DEITEL® DEVELOPER SERIES

FIFTH EDITION

Java for Programmers *with* Generative AI

- GenAI Prompt Engineering, API Calls, 600 GenAI Exercises
- ChatGPT, Gemini, Claude, Perplexity
- Multimodal: Text, Code, Images, Audio, Speech-to-Text, Text-to-Speech, Video
- Generics: Collections, Classes, Methods
- Functional Programming: Lambdas & Streams
- JavaFX: GUI, Graphics, Multimedia
- Concurrency: Parallel Streams, Virtual Threads, Structured Concurrency, Scoped Values, Concurrent Collections, Multi-Core
- Database: JDBC, SQL, SQLite
- Java Platform Module System (JPMS)
- Objects Natural: Java API, String, BigInteger, BigDecimal, Date/Time, Cryptography, ArrayList, Regex, JSON, CSV, Web Services
- JShell for Python-Like Interactivity

PAUL DEITEL • HARVEY DEITEL

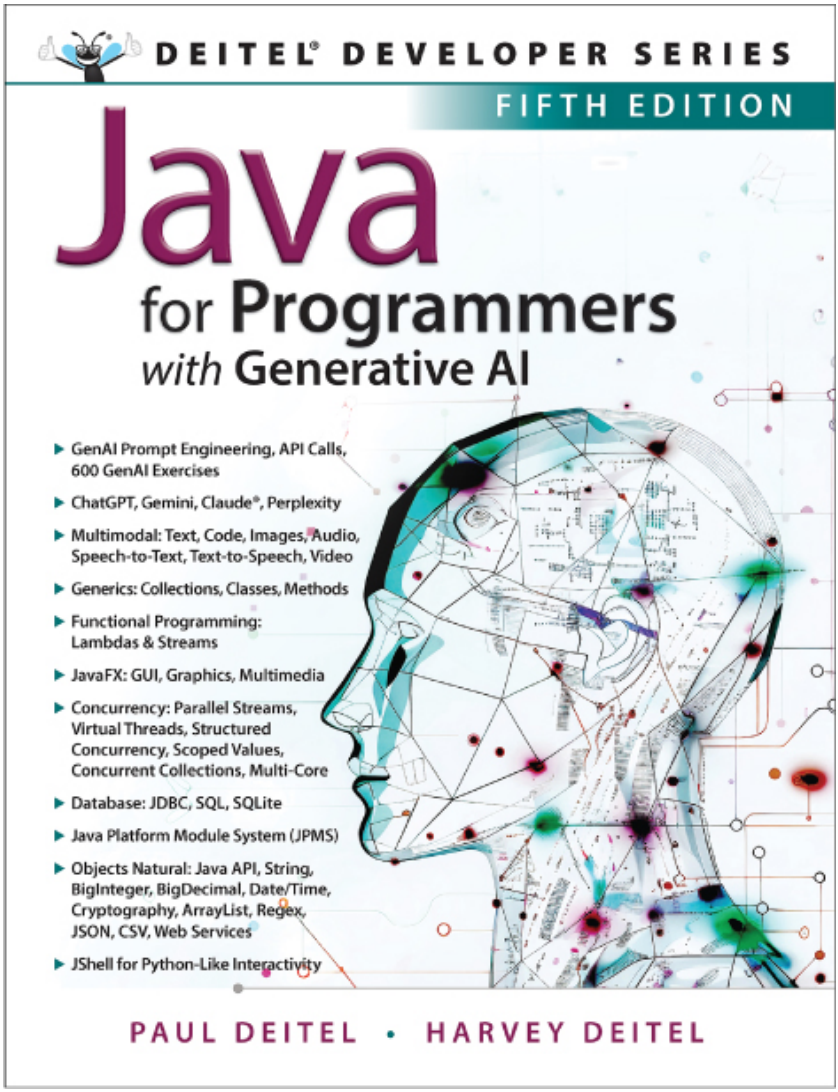
About This eBook

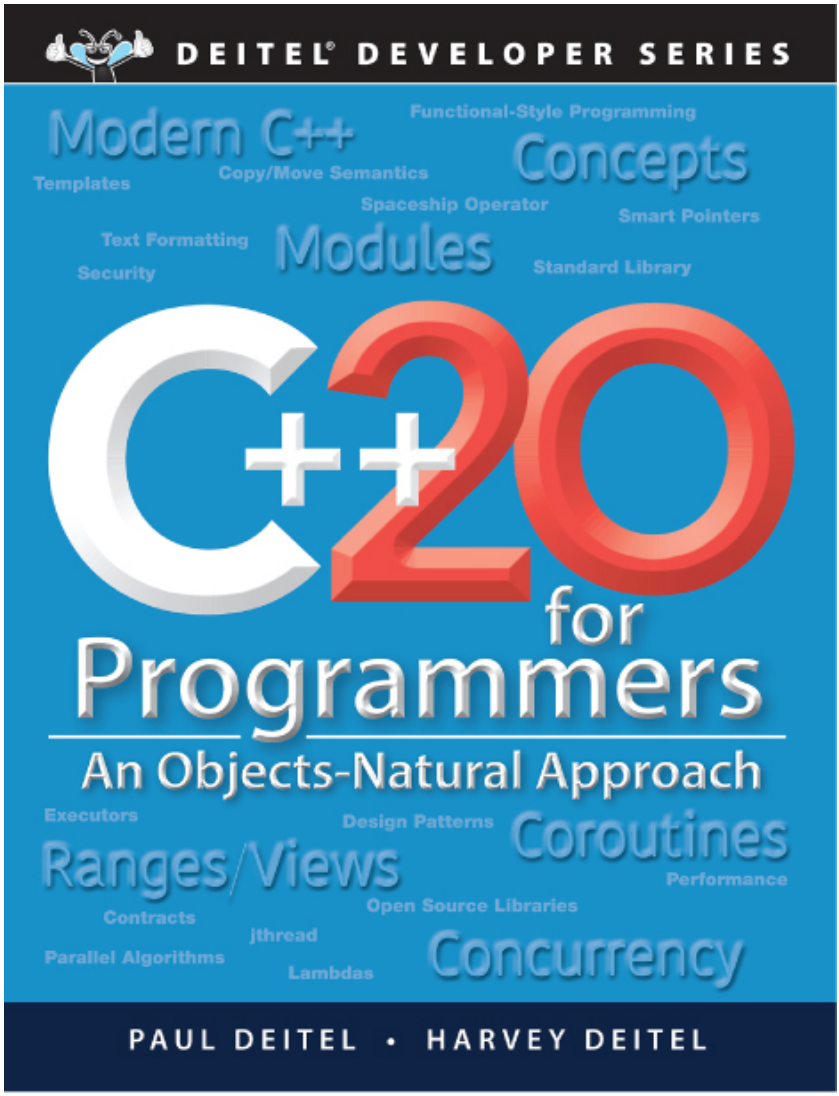
ePUB is an open, industry-standard format for eBooks. However, support of ePUB and its many features varies across reading devices and applications. Use your device or app settings to customize the presentation to your liking. Settings that you can customize often include font, font size, single or double column, landscape or portrait mode, and figures that you can click or tap to enlarge. For additional information about the settings and features on your reading device or app, visit the device manufacturer's Web site.

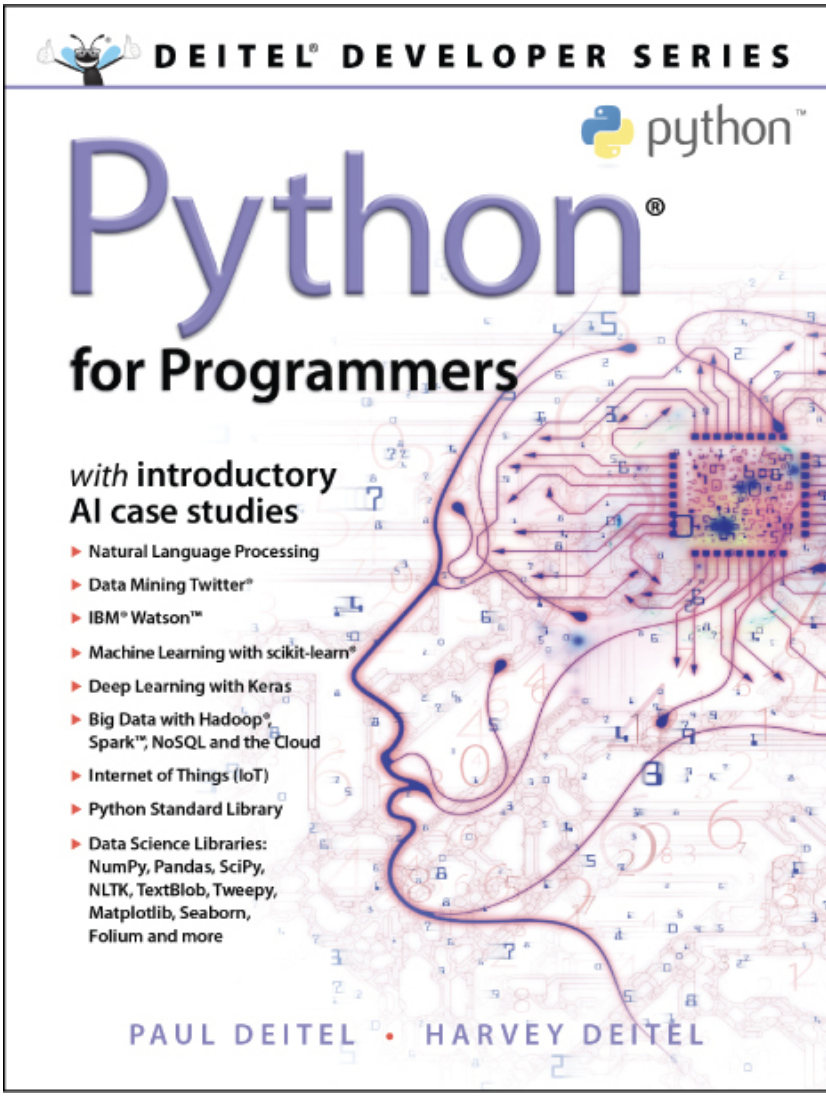
Many titles include programming code or configuration examples. To optimize the presentation of these elements, view the eBook in single-column, landscape mode and adjust the font size to the smallest setting. In addition to presenting code and configurations in the reflowable text format, we have included images of the code that mimic the presentation found in the print book; therefore, where the reflowable format may compromise the presentation of the code listing, you will see a "Click here to view code image" link. Click the link to view the print-fidelity code image. To return to the previous page viewed, click the Back button on your device or app.

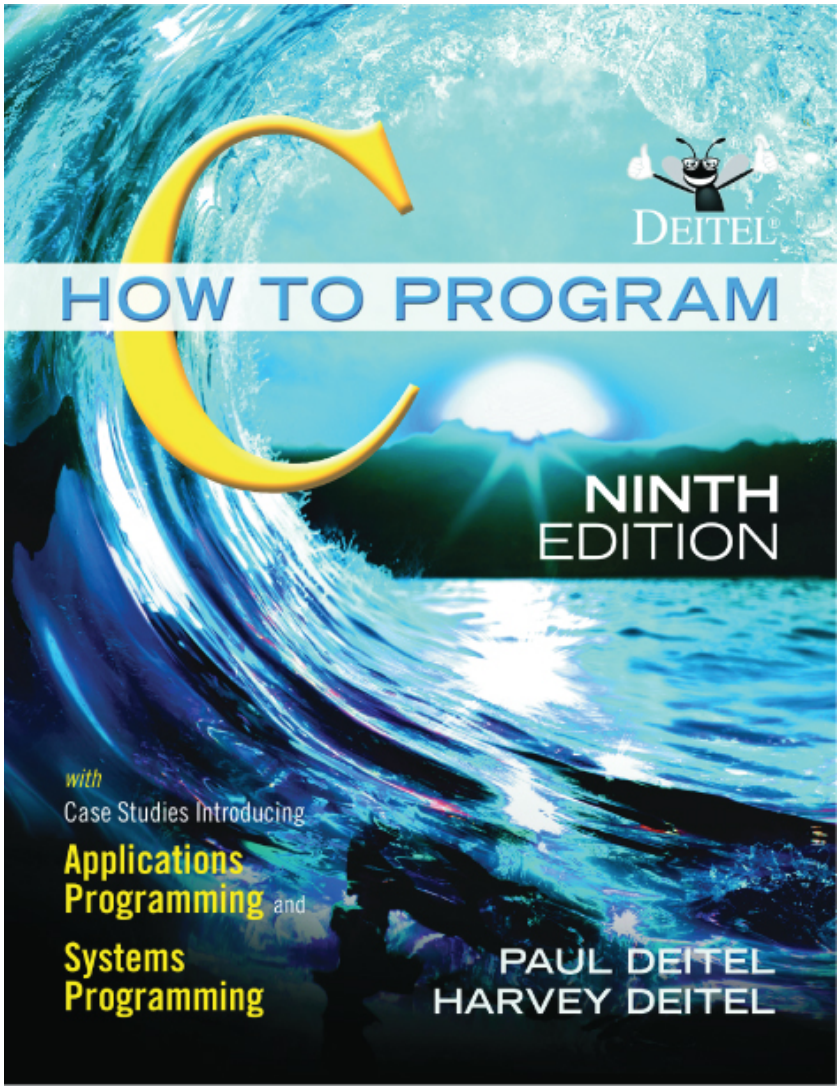
Java for Programmers *with* Generative AI

Fifth Edition


DEITEL® DEVELOPER SERIES
FIFTH EDITION
Java
for Programmers
with Generative AI
GenAI Prompt Engineering, API Calls, 600 GenAI Exercises
ChatGPT, Gemini, Claude®, Perplexity
Multimodal: Text, Code, Images, Audio, Speech-to-Text, Text-to-Speech, Video
Generics: Collections, Classes, Methods
Functional Programming: Lambdas & Streams
JavaFX: GUI, Graphics, Multimedia
Concurrency: Parallel Streams, Virtual Threads, Structured Concurrency, Scoped Values, Concurrent Collections, Multi-Core
Database: JDBC, SQL, SQLite
Java Platform Module System (JPMS)
Objects Natural: Java API, String, BigInteger, BigDecimal, Date/Time, Cryptography, ArrayList, Regex, JSON, CSV, Web Services
JShell for Python-Like Interactivity
PAUL DEITEL • HARVEY DEITEL


DEITEL® DEVELOPER SERIES
Modern C++
Functional-Style Programming
Concepts
Templates
Copy/Move Semantics
Spaceship Operator
Smart Pointers
Text Formatting
Modules
Security
Standard Library
C++20
for
Programmers
An Objects-Natural Approach
Executors
Design Patterns
Coroutines
Ranges/Views
Performance
Contracts
Open Source Libraries
jthread
Parallel Algorithms
Concurrency
Lambdas
PAUL DEITEL • HARVEY DEITEL


DEITEL® DEVELOPER SERIES
python™
Python®
for Programmers
with introductory AI case studies
Natural Language Processing
Data Mining Twitter®
IBM® Watson™
Machine Learning with scikit-learn®
Deep Learning with Keras
Big Data with Hadoop®, Spark™, NoSQL and the Cloud
Internet of Things (IoT)
Python Standard Library
Data Science Libraries: NumPy, Pandas, SciPy, NLTK, TextBlob, Tweepy, Matplotlib, Seaborn, Folium and more
PAUL DEITEL • HARVEY DEITEL



Recent Deitel Books

https://deitel.com/javafp5

https://deitel.com/cpp20fp

https://deitel.com/pythonfp

Pearson

Hoboken, New Jersey

Deitel® Developer Series

Java for Programmers *with*

Generative AI

Fifth Edition

Paul Deitel • Harvey Deitel

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The authors and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

Please contact us with concerns about any potential bias at:

https://www.pearson.com/en-us/report-bias.html

Library of Congress Control Number: On file

Copyright © 2025 Pearson Education, Inc.

All rights reserved. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, request forms, and the appropriate contacts within the Pearson Education Global Rights & Permissions Department, please visit:

https://www.pearson.com/en-us/global-permission-granting.html

Some assets have been generated by authors using the following Artificial Intelligence tools: OpenAI's ChatGPT, Google's Gemini, Anthropic's Claude and Perplexity.

Deitel and the double-thumbs-up bug are registered trademarks of Deitel & Associates, Inc.

Cover image: Kerem35/Shutterstock

Cover design by Chuti Prasertsith, Jessica Deitel, Paul Deitel, Harvey Deitel

ISBN-13: 978-0-13-757473-5
ISBN-10: 0-13-757473-8

$PrintCode

In memory of John McCarthy of Stanford University, a founder of "artificial intelligence" who coined the term and illuminated the path forward.

May this book honor your legacy by teaching how to program while collaborating with the intelligent systems you once only imagined.

Paul Deitel
Harvey Deitel

Contents

Preface

Before You Begin

1 Intro, Test-Driving a Java Application, and Generative AI

1.1 Introduction
1.2 Hardware
1.3 Java
1.4 A Brief Review of Object Orientation
1.5 Java Application Programming Interface (API) and Open-Source Libraries
1.6 Typical Java Program Development Environment
1.7 Test-Driving a Java Program with the Java Development Kit (JDK)
1.7.1 Compiling and Running the Program with `javac` and `java`
1.7.2 Running the Program's Source-Code File Directly with the `java` Command
1.8 Internet, World Wide Web, the Cloud and IoT
1.8.1 The Internet: A Network of Networks
1.8.2 The World Wide Web: Making the Internet User-Friendly
1.8.3 The Cloud
1.8.4 The Internet of Things (IoT)
1.8.5 Edge Computing

1.8.6 Mashups
1.9 Metaverse
1.9.1 Virtual Reality (VR)
1.9.2 Augmented Reality (AR)
1.9.3 Mixed Reality
1.9.4 Blockchain
1.9.5 Bitcoin and Cryptocurrency
1.9.6 Ethereum
1.9.7 Non-Fungible Tokens (NFTs)
1.9.8 Web3
1.10 Software Development Technologies
1.11 Data Analytics and Data Science
1.12 How Big Is Big Data?
1.12.1 Big-Data Analytics
1.12.2 Data Science and Big Data Are Making a Difference: Use Cases
1.13 AI—at the Intersection of Computer Science and Data Science
1.13.1 Artificial Intelligence (AI)
1.13.2 Artificial General Intelligence (AGI)
1.13.3 Artificial Intelligence Milestones
1.13.4 Machine Learning
1.13.5 Deep Learning
1.13.6 Reinforcement Learning
1.14 Generative AI
1.15 Wrap-Up

2 Intro to Java Programming

2.1 Introduction
2.2 Your First Program in Java: Printing a Line of Text
2.2.1 Compiling the Program
2.2.2 Executing the Program
2.3 Modifying Your First Program
2.4 Displaying Text with `printf`
2.5 Another Program: Adding Integers

2.6 Arithmetic
2.7 Decision Making: Equality and Relational Operators
2.8 Objects-Natural Case Study: Creating and Using Objects of the Java API's `String` Class
2.9 Wrap-Up

3 Control Statements: Part 1

3.1 Introduction
3.2 Control Structures
3.2.1 Sequence Structure
3.2.2 Selection Statements
3.2.3 Iteration Statements
3.2.4 Summary of Control Statements
3.3 `if` Single-Selection Statement
3.4 `if`...`else` Double-Selection Statement
3.4.1 Nested `if`...`else` Statements
3.4.2 Blocks
3.4.3 Conditional Operator (`?:`)
3.5 `while` Iteration Statement
3.6 Counter-Controlled Iteration
3.6.1 Implementing the Algorithm
3.6.2 Integer Division and Truncation
3.6.3 Arithmetic Overflow
3.6.4 Input Validation
3.7 Sentinel-Controlled Iteration
3.7.1 Implementing Sentinel-Controlled Iteration
3.7.2 Explicitly and Implicitly Converting Between Primitive Types
3.7.3 Formatting Floating-Point Numbers
3.8 Nested Control Statements
3.8.1 Problem Statement
3.8.2 Implementing the Program
3.9 Compound Assignment Operators
3.10 Increment and Decrement Operators

3.11 Primitive Types
3.12 Objects-Natural Case Study: Super-Sized Integers
3.13 Wrap-Up

4 Control Statements: Part 2

4.1 Introduction
4.2 Essentials of Counter-Controlled Iteration
4.3 `for` Iteration Statement
4.4 Examples Using the `for` Statement
4.5 Summing the Even Integers from 2 to 20 with the `for` Statement
4.6 Calculating Compound Interest with the `for` Statement
4.7 `do...while` Iteration Statement
4.8 `switch` Multiple-Selection Statement
4.9 `break` and `continue` Statements
4.10 Logical Operators
4.10.1 Conditional AND (`&&`) Operator
4.10.2 Conditional OR (`||`) Operator
4.10.3 Short-Circuit Evaluation of Complex Conditions
4.10.4 Boolean Logical AND (`&`) and Inclusive OR (`|`) Operators
4.10.5 Boolean Logical Exclusive OR (`^`) Operator
4.10.6 Logical Complement (`!`) Operator
4.10.7 Example: Generating Logical-Operator Truth Tables
4.11 Objects-Natural Case Study: Precise Monetary Calculations with Java API Class `BigDecimal`
4.12 Wrap-Up

5 Methods

5.1 Introduction
5.2 Declaring Methods
5.3 Notes on Declaring and Using Methods

5.4 Case Study: Die Rolling Simulation with Random-Number Generation
5.5 Case Study: A Game of Chance; Introducing `enum`s
5.6 Scope of Declarations
5.7 Method Overloading
5.8 Class `Math`: `static` Methods and Variables
5.9 Java API Packages
5.10 Method-Call Stack and Activation Records
5.11 Argument Promotion and Casting
5.12 Objects-Natural Case Study: Java Date/Time API
5.12.1 Data/Time Manipulations
5.12.2 Running the Program for a Different Locale Than Your Own
5.13 Wrap-Up

6 Arrays and `ArrayList`s

6.1 Introduction
6.2 Primitive Types vs. Reference Types
6.3 Arrays
6.4 Declaring and Creating Arrays
6.5 Creating and Initializing an Array
6.6 Array Initializers
6.7 Calculating Array Element Values
6.8 Totaling Array Elements
6.9 Intro to Visualization: Using a Bar Chart to Display Array Data Graphically
6.10 Using Array Elements as Counters
6.11 Analyzing Survey Results
6.12 Exception Handling
6.12.1 The `try` Statement
6.12.2 The `catch` Block
6.13 Enhanced `for` Statement: Totaling Array Elements
6.14 Passing Arrays to Methods
6.15 Pass-By-Value vs. Pass-By-Reference
6.16 Multidimensional Arrays

6.17 Variable-Length Argument Lists
6.18 Command-Line Arguments
6.19 Class `Arrays`
6.20 Objects-Natural Case Study: Intro to Collections and Class `ArrayList`
6.21 Wrap-Up

7 Strings, NLP and Regex: Generative AI Foundations

7.1 Introduction
7.2 Fundamentals of Characters and Strings
7.3 Class `String`
7.3.1 Creating `String` Objects
7.3.2 `String` Methods `length`, `charAt` and `getChars`
7.3.3 Comparing `String`s
7.3.4 Locating Characters and Substrings in `String`s
7.3.5 Extracting Substrings from `String`s
7.3.6 Concatenating `String`s
7.3.7 Miscellaneous `String` Methods
7.3.8 `String` Method `valueOf`
7.4 Class `StringBuilder`
7.4.1 Creating `StringBuilder` Objects
7.4.2 Methods `length`, `capacity`, `setLength` and `ensureCapacity`
7.4.3 Methods `charAt`, `setCharAt`, `getChars` and `reverse`
7.4.4 `append` Methods
7.4.5 Insertion and Deletion Methods
7.5 Class `Character`
7.5.1 `Character` Testing and Case Conversion Methods
7.5.2 Character/Digit Conversions
7.5.3 Other `Character` Methods
7.6 Tokenizing `String`s
7.7 Intro to Natural Language Processing (NLP)—at

the Root of Generative AI
7.8 Objects-Natural Case Study: Intro to Regular Expressions in NLP
7.8.1 Matching Complete `Strings` to Patterns
7.8.2 Replacing Substrings
7.8.3 Searching for Matches with Classes `Pattern` and `Matcher`
7.8.4 Simple Data Wrangling Steps Used to Prepare Text for Training NLP and Generative AI Models
7.9 Objects-Natural Security Case Study: pMa5tfEKwk59dTvC04Ft1IFQz9mEXnkfYXZwxk4ujGE=
7.10 Wrap-Up

8 Real-World Modeling with Custom Classes

8.1 Introduction
8.2 Instance Variables, *set* Methods and *get* Methods
8.2.1 `Account` Class with an Instance Variable, and *set* and *get* Methods
8.2.2 `AccountTest` Class That Creates and Uses an `Account` Object
8.2.3 Compiling and Executing a Program with Multiple Classes
8.2.4 `Account` UML Class Diagram
8.2.5 Notes on Class `AccountTest`
8.2.6 Software Engineering with `private` Instance Variables and `public` *set* and *get* Methods
8.3 Default and Explicit Instance Variable Initialization
8.4 `Account` Class: Initializing Objects with Constructors
8.4.1 Declaring an `Account` Constructor for Custom Object Initialization
8.4.2 Class `AccountTest`: Initializing New `Account` Objects
8.5 `Account` Class with a Balance
8.5.1 `Account` Class with a `BigDecimal`

balance Instance Variable
8.5.2 AccountTest Class
8.6 Case Study: Card Shuffling and Dealing Simulation
8.7 Case Study: Time Class
8.8 Controlling Access to Members
8.9 Referring to the Current Object's Members with the this Reference
8.10 Case Study: Time Class Overloaded Constructors
8.11 Default and No-Argument Constructors
8.12 Notes on *Set* and *Get* Methods
8.13 Composition
8.14 enum Types
8.15 Garbage Collection
8.16 static Class Members
8.17 static Import
8.18 final Instance Variables
8.19 Package Access
8.20 record Classes
8.20.1 Pattern Matching for switch Expressions
8.20.2 Decomposing records in switch Expressions
8.21 Wrap-Up

9 Real-World Modeling with Inheritance, Polymorphism & Interfaces

9.1 Introduction
9.2 Superclasses and Subclasses
9.3 Relationship Between Superclasses and Subclasses
9.3.1 Creating and Using a SalariedEmployee Class
9.3.2 Creating a SalariedEmployee/SalariedCommissionEmployee Inheritance Hierarchy
9.4 Class Object
9.5 Intro to Polymorphism: Polymorphic Video Game

9.6 Demonstrating Polymorphic Behavior
9.7 `abstract` Classes and Methods
9.8 Case Study: Payroll System Using Polymorphism
9.8.1 `abstract` Superclass `Employee`
9.8.2 Concrete Subclass `SalariedEmployee`
9.8.3 Concrete Subclass `CommissionEmployee`
9.8.4 Polymorphic Processing, Operator `instanceof` and Downcasting
9.9 `final` Methods and Classes
9.10 Issues with Constructors Calling Instance Methods
9.11 Creating and Using Interfaces
9.11.1 Developing a `Payable` Hierarchy
9.11.2 Interface `Payable`
9.11.3 Class `Invoice`
9.11.4 Modifying Class `Employee` to Implement Interface `Payable`
9.11.5 Using Interface `Payable` to Process `Invoice`s and `Employee`s Polymorphically
9.11.6 Some Common Java API Interfaces
9.12 Other Interface Features
9.12.1 `default` Interface Methods
9.12.2 `static` Interface Methods
9.12.3 Functional Interfaces
9.12.4 `private` Interface Methods
9.13 Program to an Interface, Not an Implementation
9.13.1 `CompensationModel` Interface
9.13.2 An `Employee` Has a `CompensationModel`—Composition and Dependency Injection
9.13.3 `CompensationModel` Implementations
9.13.4 Testing the `CompensationModel` Hierarchy
9.13.5 Dependency Injection Design Benefits
9.13.6 `interface`s vs. `abstract` Classes
9.14 `sealed` Classes and Interfaces
9.15 `private` Constructors

9.16 `protected` Members
9.17 Wrap-Up

10 Exception Handling: A Deeper Look

10.1 Introduction
10.2 Example: Divide by Zero without Exception Handling
10.3 Example: Handling `ArithmeticException`s and `InputMismatchExceptions`
10.4 Java Exception Hierarchy
10.5 Checked vs. Unchecked Exceptions
10.6 `finally` Block
10.7 Stack Unwinding and Obtaining Information from an Exception
10.8 Chained Exceptions
10.9 Declaring Custom Exceptions
10.10 Preconditions and Postconditions
10.11 Assertions
10.12 `try`-with-Resources Statement: Automatic Resource Deallocation
10.13 Unnamed Variables in `catch` Handlers
10.14 Wrap-Up

11 Files, I/O Streams, JSON Serialization & CSV Files

11.1 Introduction
11.2 Files and Streams
11.3 Using NIO Classes and Interfaces to Get File and Directory Information
11.4 Sequential Text Files
11.4.1 Creating a Sequential Text File
11.4.2 Reading Data from a Sequential Text File
11.4.3 Updating Sequential Files
11.5 Case Study: JavaScript Object Notation (JSON) Serialization
11.5.1 Serializing a `List<Account>`

11.5.2 Deserializing a `List<Account>`
11.6 Case Study: Processing a JSON Response from a Web Service
11.7 Case Study: Creating and Reading CSV Files
11.7.1 Creating a CSV File
11.7.2 Reading a CSV File
11.8 Case Study: Reading and Analyzing a CSV File Containing the *Titanic* Disaster Dataset
11.9 Objects-Natural Security Case Study: RSA Public-Key Cryptography
11.10 Wrap-Up

12 Generic Collections

12.1 Introduction
12.2 Collections Overview
12.3 Type-Wrapper Classes
12.4 Boxing and Unboxing
12.5 Lists
12.5.1 `ArrayList` and `Iterator`
12.5.2 `LinkedList`
12.5.3 Views into Collections and `Arrays` Method `asList`
12.6 `Collections` Methods
12.6.1 Method `sort`
12.6.2 Method `shuffle`
12.6.3 Methods `reverse`, `fill`, `copy`, `max` and `min`
12.6.4 Method `binarySearch`
12.6.5 Methods `addAll`, `frequency` and `disjoint`
12.7 Class `PriorityQueue` and Interface `Queue`
12.8 Hash Tables
12.9 Sets
12.10 Maps
12.11 Convenience Factory Methods for Creating Immutable Collections

12.12 Concurrent Collections
12.13 Wrap-Up

13 Generic Classes and Methods: A Deeper Look

13.1 Introduction
13.2 Motivation for Generic Methods
13.3 Generic Methods: Implementation and Compile-Time Translation
13.3.1 Generic Method `printArray`
13.3.2 Type Parameter Section of a Generic Method
13.3.3 Testing the Generic Method
13.3.4 Erasure at Compilation Time
13.4 Additional Compile-Time Translation Issues: Methods That Use a Type Parameter as the Return Type
13.5 Overloading Generic Methods
13.6 Generic Classes
13.6.1 Parameterized Types
13.6.2 Implementing a Generic `Stack` Class
13.6.3 Testing the Generic `Stack` Class
13.6.4 Refactoring the Test Program to Manipulate `Stack<E>` Objects with Generic Methods
13.7 Wildcards in Methods That Accept Type Parameters
13.7.1 Summing a `List<Number>`
13.7.2 Reimplementing Method `sum` with a Wildcard Type Argument in Its Parameter
13.8 Wrap-Up

14 Functional Programming with Lambdas & Streams

14.1 Introduction
14.2 Streams and Reduction
14.2.1 Summing the Integers from 1 through 10 with a `for` Loop
14.2.2 External Iteration with `for` Is Error

Prone
14.2.3 Summing with a Stream and Reduction
14.2.4 Internal Iteration
14.3 Mapping and Lambdas
14.3.1 Lambda Expressions
14.3.2 Lambda Syntax
14.3.3 Intermediate and Terminal Operations
14.4 Filtering
14.5 How Elements Move Through Stream Pipelines
14.6 Method References
14.6.1 Creating an `IntStream` of Random Values
14.6.2 Performing a Task on Each Stream Element with `forEach` and a Method Reference
14.6.3 Mapping Integers to String Objects with `mapToObj`
14.6.4 Concatenating Strings with `collect`
14.7 `IntStream` Operations
14.7.1 Creating an `IntStream` and Displaying Its Values
14.7.2 Terminal Operations `count`, `min`, `max`, `sum` and `average`
14.7.3 Terminal Operation `reduce`
14.7.4 Sorting `IntStream` Values
14.8 Functional Interfaces
14.9 Lambdas: A Deeper Look
14.10 `Stream<Integer>` Manipulations
14.10.1 Creating a `Stream<Integer>`, Sorting It and Collecting the Results
14.10.2 Filtering a `Stream` and Storing the Results for Later Use
14.10.3 Filtering and Sorting a `Stream` and Collecting the Results
14.10.4 Sorting Previously Collected Results
14.11 `Stream<String>` Manipulations
14.11.1 Mapping `String`s to Uppercase

14.11.2 Filtering `String`s Then Sorting Them in Case-Insensitive Ascending Order
14.11.3 Filtering `String`s Then Sorting Them in Case-Insensitive Descending Order
14.12 `Stream<Employee>` Manipulations
14.12.1 Creating and Displaying a `List<Employee>`
14.12.2 Filtering `Employee`s with Salaries in a Specified Range
14.12.3 Sorting `Employee`s By Multiple Fields
14.12.4 Mapping `Employee`s to Unique-Last-Name `String`s
14.12.5 Grouping `Employee`s By Department
14.12.6 Counting the Number of `Employee`s in Each Department
14.12.7 Summing and Averaging `Employee` Salaries
14.13 Creating a `Stream<String>` from a File
14.14 Streams of Random Values
14.15 Infinite Streams
14.16 Additional Notes on Interfaces
14.17 Wrap-Up

15 JavaFX Graphical User Interfaces: Part 1

15.1 Introduction
15.2 JavaFX Scene Builder
15.3 JavaFX Application Window Structure
15.4 Welcome Application: Displaying Text and an Image
15.4.1 Opening Scene Builder and Creating the File `Welcome.fxml`
15.4.2 Adding an Image to the Folder Containing `Welcome.fxml`
15.4.3 Creating a `VBox` Layout Container
15.4.4 Configuring the `VBox` Layout Container
15.4.5 Adding and Configuring a `Label`
15.4.6 Adding and Configuring an `ImageView`

15.4.7 Previewing the `Welcome` GUI
15.5 Tip Calculator Application: Intro to Event Handling
15.5.1 Test-Driving the **Tip Calculator**
15.5.2 Technologies Overview
15.5.3 Building the Application's GUI
15.5.4 `TipCalculator` Class
15.5.5 `TipCalculatorController` Class; Implementing an Event Handler with a Lambda
15.6 Features Covered in the Other JavaFX Chapters
15.7 Wrap-Up

16 JavaFX GUI: Part 2

16.1 Introduction
16.2 Laying Out Nodes in a Scene Graph
16.3 Painter Application: `RadioButton`s, Mouse Events and Shapes
16.3.1 Technologies Overview
16.3.2 Creating the `Painter.fxml` File
16.3.3 Building the GUI
16.3.4 `Painter` Subclass of `Application`
16.3.5 `PainterController` Class
16.4 Color Chooser Application: Property Bindings and Property Listeners
16.4.1 Technologies Overview
16.4.2 Building the GUI
16.4.3 `ColorChooser` Subclass of `Application`
16.4.4 `ColorChooserController` Class
16.5 Cover Viewer Application: Data-Driven GUIs with JavaFX Collections
16.5.1 Technologies Overview
16.5.2 Building the GUI
16.5.3 CoverViewer Subclass of `Application`
16.5.4 `CoverViewerController` Class
16.6 Cover Viewer Application: Customizing `ListView`

Cells
16.6.1 Technologies Overview
16.6.2 Copying the `CoverViewer` Application
16.6.3 `ImageTextCell` Custom `ListView` Cell Layout
16.6.4 `CoverViewerController` Class
16.7 `FileChooser` and `DirectoryChooser` Dialogs
16.8 Other JavaFX Capabilities and JavaFX Accessibility
16.9 JavaFX Updates
16.10 JavaFX Resources and Libraries
16.11 Wrap-Up

17 JavaFX Graphics and Multimedia
17.1 Introduction
17.2 Controlling Fonts with Cascading Style Sheets (CSS)
17.2.1 CSS That Styles the GUI
17.2.2 FXML That Defines the GUI—Introduction to XML Markup
17.2.3 Referencing the CSS File from FXML
17.2.4 Specifying the `VBox`'s Style Class
17.2.5 Programmatically Loading CSS
17.3 Displaying Two-Dimensional Shapes
17.3.1 Defining Two-Dimensional Shapes with FXML
17.3.2 CSS That Styles the Two-Dimensional Shapes
17.4 `Polyline`s, `Polygon`s and `Path`s
17.4.1 GUI and CSS
17.4.2 `PolyShapesController` Class
17.5 Transforms
17.6 Playing Video with `Media`, `MediaPlayer` and `MediaView`
17.6.1 VideoPlayer GUI
17.6.2 `VideoPlayerController` Class

17.7 `Transition` Animations
17.7.1 `TransitionAnimations.fxml`
17.7.2 `TransitionAnimationsController` Class
17.8 `Timeline` Animations
17.9 Frame-by-Frame Animation with `AnimationTimer`
17.10 CSS Transitions
17.11 Drawing on a `Canvas`
17.12 Three-Dimensional Shapes
17.13 FXGL: A Brief Intro to Game Programming with JavaFX
17.14 Wrap-Up

18 Concurrency: Platform Threads to Virtual Threads
18.1 Introduction
18.2 `sort/parallelSort` Timings with the Date/Time API
18.3 Sequential vs. Parallel Streams
18.4 Creating and Executing Platform Threads with the `Executor` Framework
18.5 Project Loom Overview
18.6 Creating and Executing Virtual Threads with the `Executor` Framework
18.6.1 Launching Virtual Threads with a Thread-Per-Task Executor
18.6.2 Other Ways to Launch Virtual Threads
18.7 Profiling Platform vs. Virtual Threads
18.7.1 `Future` and `Callable` Interfaces
18.7.2 Threading Approaches We'll Use
18.7.3 Launching and Timing Large Numbers of Concurrent Tasks
18.7.4 Testing the Threading Approaches
18.7.5 Performance Analysis
18.8 Structured Concurrency and Scoped Values
18.9 Thread Synchronization Overview
18.10 Producer/Consumer Relationship with `ArrayBlockingQueue`

18.10.1 Synchronization and State Dependence
18.10.2 Class `ArrayBlockingQueue`
18.10.3 Overview of the Producer/Consumer Example
18.10.4 Interface `Buffer`
18.10.5 Class `BlockingBuffer`
18.10.6 Class `BlockingBufferTest`
18.10.7 Bounded Buffers
18.10.8 Other Concurrent Collections
18.11 Multithreading in JavaFX
18.11.1 Performing Computations in a Worker Thread: Fibonacci Numbers
18.11.2 Processing Intermediate Results: Sieve of Eratosthenes
18.12 Wrap-Up

19 Building API-Based Java Generative AI Applications

19.1 Introduction
19.2 OpenAI APIs
19.2.1 Some of OpenAI's Models
19.2.2 OpenAI API Capabilities
19.2.3 Get an OpenAI Developer Account
19.2.4 OpenAI Developer API Key
19.3 Setting Up a Java Environment
19.3.1 Simple-OpenAI Open Source Library: Connecting to | OpenAI from Java
19.3.2 IntelliJ Maven-Based Java Project
19.4 Text Generation Via Chat Completions
19.4.1 Text Summarization
19.4.2 Sentiment Analysis
19.4.3 Accessible Image Descriptions
19.4.4 Language Detection and Translation
19.4.5 Code Generation
19.4.6 Named Entity Recognition (NER) and Structured Outputs
19.5 Speech Synthesis and Speech Recognition

19.5.1 English Speech-to-Text for Audio Transcription
19.5.2 Text-to-Speech
19.6 Image Generation
19.7 Video
19.7.1 Generating Video Closed Captions
19.7.2 Sora
19.7.3 Other Video GenAIs
19.8 Moderation
19.9 Class `OpenAIUtilities`
19.9.1 Builder Design Pattern
19.9.2 `package` and `import` Statements
19.9.3 Nested `record` Class `Message`
19.9.4 `SimpleOpenAI` Client Object
19.9.5 `chat` Method
19.9.6 Helper Method `makeChatMessages`
19.9.7 `describeImage` Method
19.9.8 `translate` Method
19.9.9 `chatWithStructuredOutput` Method
19.9.10 `speechToText` Method
19.9.11 `textToSpeech` Method
19.9.12 `image` Method
19.9.13 `speechToVTT` Method
19.9.14 `checkPrompt` Method
19.10 Wrap-Up

20 Accessing Databases with JDBC and SQLite

20.1 Introduction
20.2 Relational Databases
20.3 Setting Up the SQLite RDBMS
20.4 A `books` Database
20.4.1 `SELECT` Queries
20.4.2 `WHERE` Clause
20.4.3 `ORDER BY` Clause
20.4.4 Merging Data from Multiple Tables

with INNER JOIN
20.4.5 INSERT INTO Statement
20.4.6 UPDATE Statement
20.4.7 DELETE FROM Statement
20.5 Connecting to and Querying a Database with JDBC
20.5.1 Automatic Driver Discovery
20.5.2 Connecting to the Database
20.5.3 Creating a Statement for Executing Queries
20.5.4 Executing a Query
20.5.5 Processing a Query's ResultSet
20.6 Querying the books Database
20.6.1 DisplayQueryResults App's GUI
20.6.2 DisplayQueryResultsController Class
20.7 PreparedStatements
20.7.1 AddressBook App That Uses PreparedStatements
20.7.2 record Class Person
20.7.3 Class PersonQueries
20.7.4 AddressBook GUI
20.7.5 Class AddressBookController
20.8 Stored Procedures
20.9 Transaction Processing
20.10 Wrap-Up

21 Java Platform Module System
21.1 Introduction
21.2 Module Declarations
21.2.1 requires
21.2.2 requires transitive—Implied Readability
21.2.3 exports and exports...to
21.2.4 uses
21.2.5 provides...with
21.2.6 open, opens and opens...to

21.2.7 Restricted Keywords
21.3 Modularized `Welcome` App
21.3.1 `Welcome` App's Structure
21.3.2 Class `Welcome`
21.3.3 `module-info.java`
21.3.4 Module-Dependency Graph
21.3.5 Compiling a Module
21.3.6 Running an App from a Module's Exploded Folders
21.3.7 Packaging a Module into a Modular JAR File
21.3.8 Running the `Welcome` App from a Modular JAR File
21.3.9 Aside: Classpath vs. Module Path
21.4 Creating and Using a Custom Module
21.4.1 Exporting a Package for Use in Other Modules
21.4.2 Using a Class from a Package in Another Module
21.4.3 Compiling and Running the Example
21.4.4 Packaging the App into Modular JAR Files
21.4.5 Strong Encapsulation and Accessibility
21.5 Module-Dependency Graphs: A Deeper Look
21.5.1 `java.sql`
21.5.2 `java.se`
21.5.3 Error: Module Graph with a Cycle
21.6 Migrating Code to Modules
21.6.1 Unnamed Module
21.6.2 Automatic Modules
21.6.3 `jdeps` Tool for Java Dependency Analysis
21.7 Resources in Modules
21.7.1 Requiring Multiple Modules
21.7.2 Opening a Module for Reflection
21.7.3 Compiling the Module

21.7.4 Running a Modularized App
21.7.5 Module-Dependency Graph
21.8 Creating Custom Runtimes with `jlink`
21.8.1 Custom Runtime Containing Only `java.base`
21.8.2 Creating a Custom Runtime for the `Welcome` App
21.8.3 Executing the `Welcome` App Using a Custom Runtime
21.8.4 Using the Module Resolver on a Custom Runtime
21.9 Services and `ServiceLoader`
21.9.1 Service-Provider Interface
21.9.2 Loading and Consuming Service Providers
21.9.3 `uses` Module Directive and Service Consumers
21.9.4 Running the App with No Service Providers
21.9.5 Implementing a Service Provider
21.9.6 `provides...with` Module Directive and Declaring a Service Provider
21.9.7 Running the App with One Service Provider
21.9.8 Implementing a Second Service Provider
21.9.9 Running the App with Two Service Providers
21.10 Wrap-Up

22 Recursion and Big *O*

22.1 Introduction
22.2 Recursion Concepts
22.3 Recursion Example: Factorials
22.4 Recursion Example: Fibonacci Series
22.5 Recursion vs. Iteration
22.6 Towers of Hanoi

22.7 Fractals
22.7.1 Koch Curve Fractal
22.7.2 (Optional) Case Study: Lo Feather Fractal
22.7.3 (Optional) Fractal App GUI
22.7.4 (Optional) `FractalController` Class
22.8 Recursive Backtracking
22.9 Big *O* Notation
22.9.1 $O(1)$ Algorithms
22.9.2 $O(n)$ Algorithms
22.9.3 $O(n^2)$ Algorithms
22.10 Common Big *O* Notations
22.11 Wrap-Up

A Introduction to JShell for Interactive Java

A.1 Introduction
A.2 Introduction to JShell
A.2.1 Starting a JShell Session
A.2.2 Executing Statements
A.2.3 Declaring Variables Explicitly
A.2.4 Listing and Executing Prior Snippets
A.2.5 Evaluating Expressions and Declaring Variables Implicitly
A.2.6 Using Implicitly Declared Variables
A.2.7 Viewing a Variable's Value
A.2.8 Resetting a JShell Session
A.2.9 Writing Multiline Statements
A.2.10 Editing Code Snippets
A.2.11 Exiting JShell
A.3 Command-Line Input in JShell
A.4 Declaring and Using Classes
A.4.1 Creating a Class in JShell
A.4.2 Explicitly Declaring Reference-Type Variables
A.4.3 Creating Objects

A.4.4 Manipulating Objects
A.4.5 Creating a Meaningful Variable Name for an Expression
A.4.6 Saving and Opening Code-Snippet Files
A.5 Discovery with JShell Auto-Completion
A.5.1 Auto-Completing Identifiers
A.5.2 Auto-Completing JShell Commands
A.6 Exploring a Class's Members and Viewing Documentation
A.6.1 Listing Class `Math`'s `static` Members
A.6.2 Viewing a Method's Parameters
A.6.3 Viewing a Method's Documentation
A.6.4 Viewing a `public` Field's Documentation
A.6.5 Viewing a Class's Documentation
A.6.6 Viewing Method Overloads
A.6.7 Exploring Members of a Specific Object
A.7 Declaring Methods
A.7.1 Forward Referencing an Undeclared Method—Declaring Method `displayCubes`
A.7.2 Declaring a Previously Undeclared Method
A.7.3 Testing `cube` and Replacing Its Declaration
A.7.4 Testing Updated Method `cube` and Method `displayCubes`
A.8 Exceptions
A.9 Importing Classes and Adding Packages to the `CLASSPATH`
A.10 Using an External Editor
A.11 Summary of JShell Commands
A.11.1 Getting Help in JShell
A.11.2 `/edit` Command: Additional Features
A.11.3 `/reload` Command
A.11.4 `/drop` Command
A.11.5 Feedback Modes
A.11.6 Other JShell Features Configurable

with `/set`
A.12 Keyboard Shortcuts for Snippet Editing
A.13 How JShell Reinterprets Java for Interactive Use
A.14 IDE JShell Support
A.15 Wrap-Up

B Formatted Output

B.1 Introduction
B.2 Output with `printf`
B.3 Integer Formatting
B.4 Floating-Point Number Formatting
B.5 String and Character Formatting
B.6 Other Conversion Characters
B.7 Field Widths and Precisions
B.8 Formatting Flags
B.9 Argument Indices for Explicit Positioning in Format Strings
B.10 Escape Sequences
B.11 Formatting `String`s in Memory
B.12 Wrap-Up

C Number Systems

C.1 Introduction
C.2 Abbreviating Binary Numbers as Octal and Hexadecimal Numbers
C.3 Converting Octal and Hexadecimal Numbers to Binary Numbers
C.4 Converting from Binary, Octal or Hexadecimal to Decimal
C.5 Converting from Decimal to Binary, Octal or Hexadecimal
C.6 Negative Binary Numbers: Two's Complement Notation

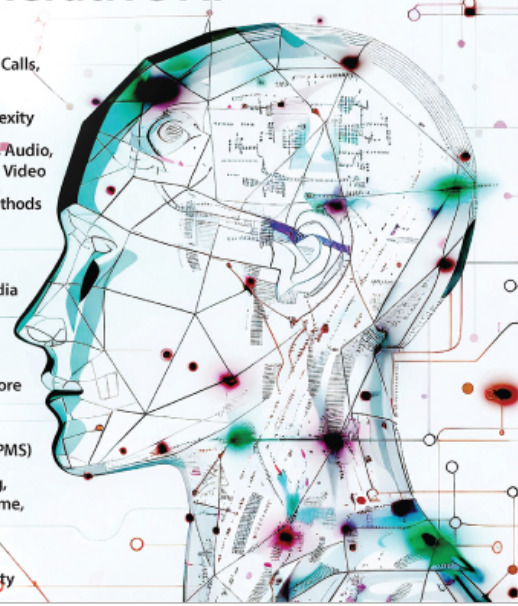
Index

DEITEL® DEVELOPER SERIES

FIFTH EDITION

Java for Programmers *with* Generative AI

- GenAI Prompt Engineering, API Calls, 600 GenAI Exercises
- ChatGPT, Gemini, Claude®, Perplexity
- Multimodal: Text, Code, Images, Audio, Speech-to-Text, Text-to-Speech, Video
- Generics: Collections, Classes, Methods
- Functional Programming: Lambdas & Streams
- JavaFX: GUI, Graphics, Multimedia
- Concurrency: Parallel Streams, Virtual Threads, Structured Concurrency, Scoped Values, Concurrent Collections, Multi-Core
- Database: JDBC, SQL, SQLite
- Java Platform Module System (JPMS)
- Objects Natural: Java API, String, BigInteger, BigDecimal, Date/Time, Cryptography, ArrayList, Regex, JSON, CSV, Web Services
- JShell for Python-Like Interactivity



PAUL DEITEL • HARVEY DEITEL

Java™ Champions

Preface

"Today, we're at the cusp of a major shift in computing. The intersection of AI and accelerated computing is set to redefine the future." —Jensen Huang, NVIDIA CEO[1]

1. Jensen Huang, NVIDIA COMPUTEX 2024 Keynote, June 2, 2024. Accessed March 2, 2025. https://www.youtube.com/watch?v=pKXDVsWZmUU.

Welcome to ***Java for Programmers with Generative AI, Fifth Edition***. In this Preface, we present the "soul of the book." This edition's key theme is **leverage**. You'll learn Java programming faster, broader and more deeply with a trio of the latest complementary pedagogic methodologies. You'll do more powerful things than with previous learning approaches and be better prepared to meet the programming and software development challenges you'll face in your professional career:

- **Powerful: We use code to teach code.** You'll learn with our traditional **live-code approach**. You'll **leverage your learning experience**, mastering Java's basics by immersing yourself in hundreds of complete working code examples with meaningful outputs.
- **More powerful: We use objects to teach objects.** New in this edition, you'll use our **objects-natural approach** to learn Java object-oriented programming. You'll **leverage your ability to write more significant programs** by using richly functional, preexisting classes from Java's extensive libraries and other open-source libraries on repository sites like `github.com`.
- **Most powerful: We use generative AI to teach generative AI.** Also new in this edition, you'll master **genAI**—the key set of artificial-intelligence technologies that **leverage your creative capabilities.** In the examples and genAI exercises, you'll use prompts and build programs that produce original text, Java code, images, audio, speech, music and even video!

A Personal Anecdote from Co-Author Harvey Deitel

In the 1970s, a friend suggested I teach an intro-to-computing course at a local college. In the segment on the hierarchy of programming-language types from **low-level machine languages** to **intermediate-level assembly languages** to **high-level languages**, each of which I programmed in, I explained that programming becomes more productive as you move up this hierarchy. You get further from the raw computer-hardware details and closer to the application domain, where you think in terms of the real-world problems you're trying to solve. Individual statements in these languages become more powerful up the hierarchy, **leveraging your ability to solve more challenging problems faster while producing better-engineered solutions**.

The students were curious, asking where all this was headed. I said, "It seems reasonable to predict that even higher-level programming technologies will appear, which will help people solve ever more challenging real-world problems faster and better. As the programming statements you write become more powerful, someday you'll be programming so close to the application domain that you'll simply **tell the computer *what* you need rather than providing the increasingly complex details of *how* to build it—and the computer will produce the software for you**!"

That was in the 1970s. Then, in the 1980s and 1990s, object-oriented programming—Java's key programming model—exploded onto the scene. Objects come from "blueprints" called classes. Almost any noun or concept can be represented by a class. There are, of course, vast numbers of those in the real world. Programmers began building extensive class libraries, **further leveraging the software-development process**. When the early versions of Java appeared in the mid-1990s, notably, they were free and accompanied by hundreds of powerful pre-built classes, making it easier for programmers to quickly build significant applications in key domains like graphics, GUI, concurrency, Internet networking, data structures, database and much more. Many of these classes had hundreds or thousands of lines of code, but to create an object of a class, a programmer typically had to write only one straightforward line of code, then use a few brief statements to make that object

"strut its stuff." This represented a **very-high-level programming capability** even closer to the problem domain.

All the while, the field of AI was evolving quickly from its roots in the 1950s. However, it wasn't until Google's 2017 research paper "Attention is All You Need"[2] defined the transformer model that the current AI boom took off. This is the root of today's large language models, which power the genAI chatbots and tools you'll use throughout this book.

2. Ashish Vaswani et al., "Attention Is All You Need." arXiv preprint, arXiv:1706.03762, 2017. Accessed January 31, 2025. https://arxiv.org/abs/1706.03762.

In November 2022, OpenAI released ChatGPT, reaching a million users in five days and 100 million in two months. **No other application had ever become so widely used that quickly while profoundly enhancing users' productivity and leveraging their ability to make computers do really interesting and powerful things**.

We have integrated genAI throughout this book. Most importantly for Java developers, genAIs remarkably can write, document, explain, debug, correct, critique, tune and enhance the performance of Java code. GenAIs are not perfect—they make mistakes and sometimes "hallucinate"—so you must monitor them carefully. We'll demonstrate these coding capabilities in the approximately 600 genAI prompting exercises and examples throughout the book. GenAI has created an **ultra-high-level programming capability** that will **leverage even further your Java learning experience and ability to produce robust, top-quality Java software quickly, conveniently and economically**.

An Extraordinary Range of Real-World Java Applications

In this book, we provide a friendly, contemporary, code-intensive, case-study-oriented introduction to Java, one of the world's most popular programming languages.[3,4] Java is popular for a wide range of applications because it's platform-independent, robust and has extensive standard and third-party class library support. It's used to build software for everything from the smallest Internet of Things (IoT) devices (like sensors connected to the Internet worldwide) to the largest cloud-based enterprise computing platforms. Some interesting uses of Java include:[5]

3. PYPL PopularitY of Programming Language Index for January 2025. Accessed January 18, 2025. https://pypl.github.io/PYPL.html.

4. Tiobe Index for January 2025. Accessed January 18, 2025. https://www.tiobe.com/tiobe-index/.

5. *ChatGPT* response to "List interesting things Java has been used for, like the Mars rover." January 15, 2025. https://chatgpt.com. We edited the response.

- Minecraft (one of the all-time most popular video games),[6]

  6. "Minecraft," Wikipedia. Last modified January 20, 2025. Accessed January 20, 2025. https://en.wikipedia.org/wiki/Minecraft.

- financial trading platforms (e.g., the LSEG Real-Time SDK[7])

  7. LSEG Developers. "Real-Time SDK for Java." Accessed January 20, 2025. https://developers.lseg.com/en/api-catalog/real-time-opnsrc/rt-sdk-java.

- Federal Aviation Administration (FAA) APIs for flight and airport information,[8]

  8. Federal Aviation Administration, GitHub Repository. Accessed January 20, 2025, https://github.com/Federal-Aviation-Administration.

- Android mobile app development[9]—Android is the leading mobile device operating system, with a 73.5% market share worldwide,[10]

  9. "Welcome to Android Developers." Accessed January 15, 2025. https://developer.android.com/.

  10. "Mobile Operating System Market Share Worldwide." StatCounter. Accessed January 20, 2025, https://gs.statcounter.com/os-market-share/mobile/worldwide.

- NASA Mars rover software[11] and

  11. Ross, Philip. "Java Runs Remote-Controlled Mars Rover." CNET. Java 16, 2004. Accessed January 20, 2025. https://www.cnet.com/tech/tech-industry/java-runs-remote-controlled-mars-rover/.

- tools for analyzing genetic DNA and RNA sequencing data.[12]

  12. Broad Institute. "Getting Started with GATK4." GATK Documentation. Updated July 20, 2024. Accessed January 20, 2025. https://gatk.broadinstitute.org/hc/en-us/articles/360036194592-Getting-started-with-GATK4.

Other Java uses listed by the popular genAI chatbots ChatGPT,[13]

Gemini,[14] Claude[15] and Perplexity that you'll use throughout this book[16] include:

- desktop GUI applications (as we do beginning with JavaFX in Chapters 15–17),
- cross-platform 2D and 3D game-development libraries (such as FXGL, which we will cover in a blog post),
- embedded systems (such as smart cards, Blu-ray players, cable set-top boxes, printers and automotive "infotainment" systems),
- scientific and educational software,
- web-based applications (such as the Spring and Spring Boot frameworks),
- high-performance server-side software,
- enterprise solutions for banking and finance,
- large-scale distributed systems,
- robotics, and
- big data technologies (like Hadoop that store and process vast amounts of data).

13. *ChatGPT* response to "Give me 100 words on what kinds of apps Java is popular for." January 16, 2025. https://chatgpt.com. We edited the response.
14. *Gemini* response to "Give me 100 words on what kinds of apps Java is popular for." January 16, 2025. https://gemini.google.com/app. We edited the response.
15. *Claude* response to "Give me 100 words on what kinds of apps Java is popular for." January 16, 2025. https://claude.ai. We edited the response.
16. *Perplexity* response to "Give me 100 words on what kinds of apps Java is popular for." January 16, 2025. https://perplexity.ai. We edited the response.

Target Audiences

The book's modular architecture (see the **"High-Level Overview"** below) makes it appropriate for several audiences:

- Introductory and intermediate professional Java

programming training courses based on the curriculum recommendations of the **ACM** and **IEEE** professional societies.[17]

17. ACM/IEEE (Assoc. Comput. Mach./Inst. Electr. Electron. Eng.). 2023. *Computer Science Curricula 2023* (New York: ACM). Accessed January 18, 2025. https://csed.acm.org/wp-content/uploads/2023/03/Version-Beta-v2.pdf.

- Programmers experienced in other languages who must learn Java for use in upcoming projects.
- Programmers who know some Java and need to learn the latest Java features and idioms to prepare for upcoming projects.

It will also help you prepare for most topics covered by the **Oracle Java SE (Standard Edition) Developer Professional certification exam.**[18,19]

18. Jeanne Boyarsky and Scott Selikoff, *OCP Oracle Certified Professional Java SE 21 Developer Study Guide* (New York: Wiley, 2024). [Jeanne Boyarsky was a technical reviewer on *Java for Programmers with Generative AI, 5/e*.]

19. "Exam: Java SE 21 Developer Professional (1Z0-830)." Accessed January 18, 2025. https://mylearn.oracle.com/ou/exam/java-se-21-developer-professional-1z0-830/40805/139080/220555.

Live-Code Approach

At the heart of the book is the Deitel signature **live-code approach**. Rather than code snippets, you'll learn Java hands-on from 200+ fully coded, real-world examples and case studies with live outputs drawn from computer science, data science, AI and other fields. Read the Before You Begin section that follows this Preface to learn how to download the code examples and set up your Windows or macOS computer to run them.

Chapter 1's Test-Drive (Section 1.7) shows how to compile and run the code examples with the free, open-source OpenJDK version of the Java Development Kit. **Executing each program in parallel with reading the text will make your Java learning experience "come alive."** We also provide test-drives of the following popular Java developer tools at https://deitel.com/javafp5:

- JetBrains IntelliJ IDEA Community Edition

- The Eclipse IDE for Java Developers
- Microsoft Visual Studio Code

Java for Programmers with Generative AI, 5/e

High-Level Overview

Programming Paradigms

Procedural (Chapters 2–7), **object-oriented** (Chapters 8–10), **generic** (Chapters 12 and 13), **functional** (Chapter 14), **concurrent** (Chapter 18) and **genAI-enhanced** (Chapter 19).

Part 1: Introduction

Chapter 1, Intro, Test-Driving a Java Application, and Generative AI—Internet and Web. "Cloud." Internet of Things (IoT). Metaverse. Intro to AI/genAI. Test-driving the Java Development Kit (JDK). **27 genAI prompt exercises**.

Chapter 2, Intro to Java Programming—Java fundamentals. Input/output. Primitive types. Arithmetic. Decision making. **Objects-natural (ON) case study: `String` class. 12 genAI prompt exercises.**

Part 2: Additional Programming Fundamentals

Chapter 3, Control Statements: Part 1—`if`, `if/else`, `while` statements. Counter-controlled/sentinel-controlled iteration. Nested control statements. **ON case study: `BigInteger` for supersized integers. 16 genAI prompt exercises.**

Chapter 4, Control Statements: Part 2—`for`, `do/while`, `switch`, `break`, `continue` statements. Logical operators. **ON case study: `BigDecimal` for precise monetary calculations. 13 genAI prompts.**

Chapter 5, Methods—Custom methods. Random numbers. Simulation. **ON case study: Date/Time API. 16 genAI prompt exercises.**

Chapter 6, Arrays and `ArrayList`s—Data structures. Built-in arrays. **ON case study: `ArrayList` for dynamically resizable arrays. 28 genAI prompt exercises.**

Chapter 7, Strings, NLP and Regex: Generative AI Foundations—`String`, `Character`, `StringBuilder` classes. Intro to natural language processing (NLP). **ON case studies: 1. Regular expressions for pattern matching in text, 2. Securing data with AES private-key cryptography. 22 genAI prompt exercises.**

Part 3: Object-Oriented Programming

Chapter 8, Real-World Modeling with Custom Classes—Crafting valuable classes. `Account` class case study. Card-shuffling-and-dealing simulation. `Time` class case study. Controlling class-member access. Constructors. *Set/get* methods. Data validation. Throwing exceptions. `static` and `final` class members. `record` classes. **33 genAI prompt exercises.**

Chapter 9, Real-World Modeling with Inheritance, Polymorphism & Interfaces—Inheritance hierarchies. Runtime polymorphism. Inheritance *is-a* relationship vs. composition *has-a* relationship. Interfaces. Programming to an interface, not an implementation. Dependency injection. `sealed` classes and interfaces. **31 genAI prompt exercises.**

Chapter 10, Exception Handling: A Deeper Look—`try` statement to catch and handle exceptions. `Throwable` hierarchy. Checked vs. unchecked exceptions. `try`-with-resources statement. **30 genAI prompt exercises.**

Chapter 11, Files, I/O Streams, JSON Serialization & CSV Files—Data persistence. Text vs. binary files. NIO classes. Retrieving file/directory info. `Formatter` class. JSON (JavaScript Object Notation) serialization. Invoke an OpenWeatherMap web service with `java.net.http` features. CSV (comma-separated values) file format. Titanic disaster CSV dataset and basic data analytics. **ON case study: Securing data and protecting user privacy with RSA public-key cryptography. 38 genAI prompt exercises.**

Part 4: Data Structures, Generic Collections, Lambdas and Streams

Chapter 12, Generic Collections—Prepackaged data structures from the Java collections framework. Pre-built generic data structures. **30 genAI prompt exercises.**

Chapter 13, Generic Classes and Methods: A Deeper Look—Implement a custom generic method and a custom generic class. Compile-time type safety. **24 genAI prompt exercises.**

Chapter 14, Functional Programming with Lambdas & Streams

—Use lambdas and stream pipelines to write certain kinds of programs faster, simpler, more concisely and hopefully with fewer bugs than previous techniques. Focus on immutability. Chapter 18, Concurrency: Platform Threads to Virtual Threads, demonstrates parallelizing stream pipelines to enhance performance on multi-core architectures. **37 genAI prompt exercises.** [Chapter 22 can be covered here.]

Part 5: JavaFX Graphical User Interfaces, Graphics and Multimedia

Chapter 15, JavaFX Graphical User Interfaces: Part 1—Scene Builder for simple drag-and-drop GUI design. Layouts. Controls. Event handling. Welcome app. Tip Calculator app. **13 genAI prompt exercises.**

Chapter 16, JavaFX GUI: Part 2—Additional layouts and controls. Mouse, `RadioButton` and property-change events. Data binding. Customizing a control's appearance. `FileChooser` and `DirectoryChooser` dialogs. **21 genAI prompt exercises.**

Chapter 17, JavaFX Graphics and Multimedia—Cascading Style Sheets (CSS) for customizing JavaFX nodes' appearance and text fonts. 2D and 3D shapes. Move, rotate and scale node transformations. Display, play and pause video. Incrementally change properties with `Transition` and `Timeline` animations. Frame-by-frame animations with `AnimationTimer`. CSS transitions for simplified animation effects. `Canvas` for drawing with the mouse. **31 genAI prompt exercises.**

Part 6: Advanced Topics

Chapter 18, Concurrency: Platform Threads to Virtual Threads—Multi-core programming. Create and manage multiple tasks. Performance case studies. Profile sequential vs. parallel sorting with Date/Time APIs. Prepackaged parallel algorithms. Classic producer–consumer relationship. Easier-to-use, less error-prone, higher-level concurrency features. Project Loom: Lightweight virtual threads, structured concurrency, scoped values. **51 genAI prompt exercises**.

Chapter 19, Building API-Based Java Generative AI Applications—Multimodal genAIs that understand text, code, image, speech and video inputs and generate text, code, images, speech and video. Work with OpenAI APIs to create apps that summarize text, analyze text for sentiment, create accessible descriptions of images, detect a text's language and translate text

among languages, generate Java code, perform named entity recognition on text and obtain structured JSON outputs, transcribe speech to text, synthesize speech from text, generate closed captions for a video and more. **21 genAI prompt exercises and 73 API-based programming projects.**

Chapter 20, Accessing Databases with JDBC and SQLite—Storing persistent data in databases. SQLite database management system. Structured Query Language (SQL) CRUD (create, read, update, delete) operations. JDBC API. Connecting to a database. Retrieve data from a database. JavaFX `TableView`. JDBC `PreparedStatement`s. **19 genAI prompt exercises**.

Chapter 21, Java Platform Module System—Create custom packages and modules. Declare module dependencies. Specify which packages a module exports. Define the services a module offers or consumes. Reflection and enabling reflective access. Bundling resources with modules. **31 genAI prompt exercises**.

Chapter 22, Recursion and Big O—Recursive and iterative factorial calculations. Recursive Fibonacci calculations. JavaFX app that recursively creates a "feather fractal." Text-based algorithm visualizations. Rich selection of recursion exercises. **11 genAI prompt exercises.** [Can be covered at the end of Part 4: Data Structures.]

Appendices

Appendix A, Introduction to JShell for Interactive Java—Java's friendly, command-line REPL (read-evaluate-print-loop) for exploration, discovery and experimentation. Like having Python's interactivity in Java. Many sections can be read in conjunction with the book's early chapters.

Appendix B, Formatted Output—formatting features for output and for `String`s in memory. Summarizes the formatting features we discuss throughout the book and introduces additional capabilities.

Appendix C, Number Systems—Introduces binary (base 2), octal (base 8), decimal (base 10) and hexadecimal (base 16) number systems.

Possible Topics for the Deitel.com Blog

Sequenced collections. Flexible constructor bodies. Bit manipulation. Labeled `break` and `continue` statements. JavaFX `Subscription` API. Intro to basic JavaFX game development with FXGL (FX Game Library), animation, collision detection, particle

effects and more. We'll link our Java-related blog posts to the book's webpage at https://deitel.com/javafp5

"Objects-Natural" Approach

Object-oriented programming books have traditionally used a "late objects" or an "early objects" teaching approach. What's really "late" or "early" in these books is not "objects." It's teaching how to develop custom classes—the "blueprints" from which objects are built. We've written books using both approaches in popular object-oriented programming languages.

What Is "Objects Natural"?

As we wrote our textbook *Intro to Python for Computer Science and Data Science: Learning to Program with AI, Big Data and the Cloud*,[20] we noticed that although our presentation fits the "late objects" model, it is actually something more—and that something is special. We call it the **"objects-natural approach,"** and we're now using it in each object-oriented programming language textbook and professional book we write.

20. Deitel, Paul, and Harvey Deitel. *Intro to Python for Computer Science and Data Science: Learning to Program with AI, Big Data and the Cloud*. Upper Saddle River, NJ: Pearson, 2020.

Like "late objects," our objects-natural approach begins with programming fundamentals such as data types, variables, operators, control statements, methods, arrays and strings in the early chapters—all before you develop your own *custom* classes. However, with the objects-natural approach, you'll get lots of practice using powerful *existing* classes that do significant things, quickly creating objects of those classes (typically with one line of code) and telling them to "strut their stuff" with a minimal number of simple Java statements. This is one of the most compelling aspects of working with a mature object-oriented language like Java. And you'll do this long before you create and use custom Java classes in Chapter 8 and inheritance, polymorphism and interfaces in Chapter 9.

An Abundance of Free Classes

We emphasize using the massive number of valuable free classes in

the Java ecosystem. These typically come from:

- the **Java Application Programming Interface (API)—Java's standard library**—and
- free third-party Java libraries, often created by the open-source community.

We encourage you to view lots of Java code on sites like GitHub. Reading other programmers' code is a great way to learn.

Generative AI (GenAI) Approach[21]

21. *ChatGPT* (https://chatgpt.com), *Gemini* (https://gemini.google.com), *Claude* (https://claude.ai) and *Perplexity* (https://perplexity.ai) responses to the prompt, "The following is a list of miscellaneous notes on how we used genAIs when working on our new Java programming textbook and how our readers will use genAI while reading the book. Turn these notes into useful flowing text, reorganizing the points as necessary and removing redundancy." January 1, 2025. To demonstrate the power of genAIs, we followed the preceding prompt with a lengthy (and sometimes repetitive) list of our raw author notes for this section, asking the genAIs to organize our notes into smooth, flowing paragraphs. We then edited the final presentation.

Leveraging genAI was integral in developing *Java for Programmers with Generative AI, 5/e*. It will also be essential to how you interact with the material. Here, we outline our approach to integrating genAI throughout the book, its potential to profoundly enhance Java education, and **how to leverage these tools effectively while maintaining a cautious and critical mindset**.

While writing the book, we integrated genAI not as a replacement for our work as authors but as a powerful enhancement tool. Similarly, you should use genAI to augment your learning and software-development skills, not to replace them. **Our experience with genAI has been both enlightening and entertaining—it's like having a diverse team of experts available to assist you 24/7, though one that requires careful supervision.**

Two Approaches to Using GenAI

We integrate genAI using two approaches:

- **Prompt engineering:** Outside Chapter 19, we focus on prompt engineering to optimize the interaction with genAI tools. Think of this as learning a new and different type of programming language. With prompt engineering, you craft precise instructions or queries to guide genAIs in producing the required responses. You can also iteratively tune your prompts, refining and adjusting them to help the genAIs produce more accurate and relevant results—**the better your prompts, the better the results**. It takes lots of experience to learn how to create great prompts—you'll get that experience in this book.
- **Java code using AI APIs plus prompt engineering:** Chapter 19 combines prompt engineering and API-based code development, emphasizing how genAI can help you build and tune complete Java applications. We use the OpenAI APIs.

Chapter 19's examples and exercises dive deeper into using **genAI for code generation and analysis** as we build API-based Java genAI applications. You'll learn to craft effective prompts to produce code snippets, debug and tune your programs, and create complete Java applications.

GenAI Tools We Used

Our work relied on many genAIs, including four primary genAI chatbots:

- OpenAI ChatGPT
- Google Gemini
- Anthropic Claude
- Perplexity

We used them all for each task because they produced some overlapping and some different results—each added value. In fact, the same genAI often yielded different results when we gave it the same prompt, even if only moments later. Each has unique strengths and weaknesses. GenAIs constantly evolve, so capabilities and limitations will change over time, most likely improving dramatically.

We also used **OpenAI's Dall-E** for **image generation** (Dall-E is

now integrated into ChatGPT) and **Sora** for **video generation**. Many genAIs offer free trials or tiers, which can give you good results but might be less capable than their paid counterparts.

Role of GenAIs in Education

GenAIs will play a crucial role in professional and college education in general and especially in programming-language education. They serve as an always-available team of virtual experts to assist you, accessible at little or no cost.

Organizations and educational institutions remain divided on whether to allow genAI usage. Some embrace it as we do in this book. Some have banned it. Others permit it, with safeguards. We encourage you to become familiar with these tools, which are likely to be indispensable throughout your career. However, comply carefully with your organization's or institution's policies.

GenAIs for Enhancing Learning Experiences

GenAIs can enhance learning experiences by:

- Providing multiple perspectives on complex concepts.
- Offering interactive debugging assistance.
- Helping identify areas for review when concepts are unclear.
- Supporting various learning styles through different explanatory approaches.
- Offering personalized feedback on the code you write.

We encourage you to do the work, using genAIs to enhance your learning experience. Our Generative AI exercise sections help you:

- Learn prompt engineering—crafting and refining prompts to achieve optimal results.
- Learn what genAIs can do by trying lots of prompts.
- Learn the level of detail with which each genAI responds to prompts.
- Discover new ways to solve problems.
- Compare the strengths and weaknesses of different genAIs.
- Understand that genAIs make mistakes and sometimes

hallucinate, and how to detect and address those errors.

- Be aware that genAIs can produce inaccurate results.
- Get clarification on points raised in the book.
- Enhance your knowledge when you want more detail.

How We Used GenAIs

We integrated genAI into our authoring efforts—not to write the manuscript (except by design in cited instances) or code examples but to enhance research, development, tuning the writing and proofreading. We used genAIs to:

- Refine our genAI prompting skills to get better results.
- Brainstorm.
- Extract key points from this book's content and use them to draft PowerPoint slides for instructors to prepare and deliver their classes.
- Verify our content.
- Check that we used current Java programming idioms and clean code[22] guidelines appropriate for a book at this level.

  22. Martin, Robert C. *Clean Code: A Handbook of Agile Software Craftsmanship*. Upper Saddle River, NJ: Addison-Wesley, 2009.
- Check our grammar and writing smoothness and clarity.
- Generate multiple-choice question drafts from our content.
- Draft exercise solutions using only the techniques we presented to a given point in the book.
- Draft marketing text directly from our content.
- Draft blog posts directly from our content.
- Refactor our code to conform to our coding conventions, such as code indentation and comment relevance and density.
- Refactor code (e.g., renaming variables, methods and classes, extracting repetitive code into separate functions, etc.).

Effective prompt engineering was key in most cases, and revisiting an ongoing chat with a carefully tuned prompt often led to dramatically improved results.

Guidelines for GenAI Users

We encourage you to use genAIs as a learning tool rather than just an answer generator. Just as programming mastery requires understanding language and library capabilities, effective genAI use requires developing skills with different prompts. You should:

- Keep track of your prompts and the quality of their results. This is crucial for learning what works and what doesn't. This will also help you identify the nuances of different genAI platforms. You'll build a repertoire of valuable prompts and anticipated results that will leverage future programming efforts.
- Experiment with multiple genAIs—including new ones as they appear—and compare the results to understand their diverse capabilities.
- Experiment with different prompting strategies to see which ones help you achieve your desired outcomes. Different programming challenges can require different prompting strategies.
- **If genAI results are unsatisfactory, tune your prompt and try again**. For example, you'll see that some genAIs emphasize producing smooth, professional quality writing, while others dig deeper into the subject matter.
- Use genAIs for **discovery**—finding new ideas, insights and patterns that might not initially be apparent. GenAIs offer wide-ranging perspectives, analyzing vast amounts of data and synthesizing information into actionable outputs that can enhance creativity or inform decisions. GenAI discovery can enhance your creativity and uncover possibilities beyond your expertise, helping you grow your knowledge and wisdom.[23]

23. *ChatGPT* response to "What does it mean to use generative AIs for discovery?" January 18, 2025. https://chatgpt.com. We edited the response.

Hallucinations and Errors

GenAIs have limitations. They've been known to "hallucinate," confidently providing fabricated answers. When you ask them to perform tasks, sometimes the results contain errors, such as Java code that does not work. We've included successful and

unsuccessful genAI interactions in the book. For instance, while genAIs helped us enhance our Number Systems Appendix with innovative applications of hexadecimal, we also encountered situations where they struggled with seemingly straightforward tasks. As early as Chapter 2, we encountered genAI failures when asking them to write simple Java statements. We deliberately included some of these failure cases as valuable teaching moments, demonstrating that genAIs are fallible, so you must use them cautiously. **This underscores the importance of maintaining human oversight and verification, especially in crucial business-critical and mission-critical situations.**

Why We Use Some Paid Services

While free or free-tier genAIs are valuable, paid services often offer more robust capabilities, availability and reliability. Many genAIs are paid services due to the sheer cost of training the large language models that provide the genAIs' powerful capabilities. For example, training OpenAI's GPT-4 and Google's Gemini cost $78 million and $191 million, respectively—not including the salaries of the people developing the large language models and other overhead costs.[24] We encourage you to use free or low-cost tools when available. You can even ask genAIs to suggest the best free versions. However, professional developers will often choose paid tools to get the best results.

24. Stanford Institute for Human-Centered Artificial Intelligence (HAI), AI Index Report 2024, May 2024. Accessed March 8, 2025. https://hai-production.s3.amazonaws.com/files/hai_ai-index-report-2024-smaller2.pdf.

What We're Trying to Accomplish

We aim to prepare you for a future in which genAIs will be indispensable in your personal life and professional career. By integrating genAI examples and exercises throughout the book, we aim to help you become comfortable using these tools and understand their advantages and pitfalls. Developing a healthy sense of caution while exploring genAI capabilities will empower you to use them effectively.

Chapter 19's API-Based, Complete, Live-Code Java GenAI App Case Studies

Chapter 19 presents the following fully implemented code examples

that programmatically interact with OpenAI's APIs:

- **Text Summarization (**Section 19.4.1**):** Asks the OpenAI chat API for a summary abstract paragraph of a Deitel video transcript and a list of the transcript's key points. Text responses like these are known as **chat completions**.
- **Sentiment Analysis** (Section 19.4.2): Asks the OpenAI chat API for the sentiment of a transcript and an explanation of how the genAI came to its conclusion.
- **Accessible Image Descriptions** (Section 19.4.3): This multimodal application uploads images and asks the OpenAI chat API for detailed descriptions of the `for`-loop UML diagram in Section 4.3 and a scenic Aruba beach photo.
- **Language Detection and Translation (**Section 19.4.4**):** Asks the OpenAI chat API to translate text to a specified target language—it **autodetects the source language**. We translate text from English to Spanish and Japanese, then from Spanish and Japanese back to English. Many of today's genAIs can translate among scores of languages.
- **Java Code Generation (**Section 19.4.5**):** Asks the OpenAI chat API to generate Java code for a simulation summarizing millions of six-sided die rolls and displaying the results.
- **Named-Entity Recognition and Structured Outputs (**Section 19.4.6**):** Demonstrates producing a structured JSON response in your specified format. This app performs a natural language processing (NLP) task called named entity recognition (NER), which attempts to locate and categorize text mentions like dates, times, quantities, places, people, things, organizations, etc. To make the results easier to process, we provide the API with the exact JSON response format we wish to receive, then use the **Jackson open-source library** (Chapter 11) to process the results.
- **Speech-to-Text** (Section 19.5.1): This multimodal application (speech audio and text) uploads a speech audio file to the **OpenAI Whisper model**, which transcribes it to text and returns the text transcription.
- **Text-to-Speech** (Section 19.5.2): This multimodal application (text and speech audio) uploads text to the **OpenAI TTS-1 model** to synthesize speech from the text. This example converts English, Spanish and Japanese text samples into speech using the same voice.

- **Image Generation** (Section 19.6): Demonstrates code that asks the **Dall-E APIs** to generate images of a Havanese dog in the styles of a Japanese anime character in neon colors against a black background, Vincent Van Gogh and Leonardo DaVinci.
- **Creating Closed Captions for a Video** (Section 19.7.1): This multimodal accessibility application (speech audio, text and video playback) uploads an audio track from a video and transcribes it into a JSON format containing timestamps and corresponding transcribed text. The app then requests a chat completion that generates **closed captions in VTT format**.[25] We show VTT-formatted text and screen captures displaying captions over the corresponding video.

  25. "WebVTT." Wikipedia. Last modified December 31, 2024. Accessed March 1, 2025. https://en.wikipedia.org/wiki/WebVTT.

- **Moderation** (Section 19.8): Asks the **OpenAI moderation API** to evaluate text prompts for harmful or inappropriate text, such as harassment, hate speech, violence, self-harm and sexually explicit content.

GenAI Prompt Exercises

We fed the complete list of all the book's approximately 600 genAI exercises (a 100+ page PDF) to ChatGPT, Gemini, Claude and Perplexity, asking them to categorize the kinds of things we do in those exercises. Next, we fed all their categorized lists to the four genAIs, asking them to summarize the summaries, and we chose the best one—Claude in this case:

- **Code Generation and Implementation—**Writing new Java programs from specifications. Implementing specific features, algorithms and APIs. Creating test programs and practical applications. Generating solutions for both basic and advanced programming tasks.
- **Code Refactoring and Enhancement—**Modernizing code with current Java features (streams, enhanced loops). Improving code structure, readability, and maintainability. Converting between different approaches while maintaining functionality. Improving performance.
- **Educational Content—**Creating tutorials, exercises, and learning materials. Explaining complex concepts

(polymorphism, immutability, etc.). Developing beginner-friendly programming exercises. Writing comprehensive documentation and guides.

- **Technical Analysis—**Analyzing code behavior and feature implementations. Comparing different approaches, tools, and frameworks. Evaluating trade-offs in design decisions. Breaking down complex technical concepts.
- **Best Practices and Standards—**Implementing coding standards and design patterns. Addressing security considerations. Optimizing performance. Following Java development best practices.
- **Technology Evaluation—**Comparing libraries, tools, and frameworks. Assessing the pros and cons of different approaches. Making informed technology choices. Exploring new features and updates.
- **Debugging and Error Handling—**Finding and fixing syntax and logical errors. Implementing exception handling. Improving fault tolerance. Preventing common pitfalls.
- **API and Library Integration—**Working with Java APIs and external libraries. Understanding API features and capabilities. Implementing integration techniques. Creating API documentation and tutorials.
- **Real-world Applications—**Developing practical use cases and industry applications. Creating interactive applications (GUIs, games, multimedia). Implementing real-world scenarios. Building sample projects.
- **Performance Optimization—**Analyzing and improving performance. Optimizing resource usage. Conducting benchmarks. Implementing efficiency improvements.
- **Creative Development—**Building multimodal applications. Creating visualizations. Generating test scenarios and sample data. Developing unique use cases.

GenAI API-Based Java Programming Exercises

Chapter 19, Building API-Based Java Generative AI Applications, suggests challenging project exercises like creating genAI multimedia apps that can debate one another and using genAI to build and solve crossword puzzles. We fed Chapter 19's 90+ exercises into the genAIs, asking for a categorized summary of

them, then summarized the summaries. Here's what they produced:

- **Multimodal Applications—**Combining text, image, audio, and video capabilities. Creating integrated experiences like interactive books. Developing multimedia educational content. Building comprehensive tools that leverage multiple AI modalities.
- **Text-Based Applications—**Document processing (indexing, summarization, exploration). Creative writing (story generation, poetry, debates). Language tools (translation, tone rewriting). Professional document creation (resumes, presentations). Structured outputs.
- **Image Processing Applications—**Generative art and design (logos, fashion, floor plans). Technical visualization (UML diagrams). Image analysis and recognition.
- **Audio and Music Applications—**Speech processing (transcription, voice cloning). Music generation (MIDI, Magenta AI). Multilingual audio applications. Podcast and audio content analysis.
- **Educational Tools—**Programming tutors (Java, coding exercises). Subject-specific learning aids (math, arithmetic). Course content creation. Interactive educational experiences.
- **Gaming and Puzzle Applications—**Puzzle generators and solvers (Crosswords, Word Search). Interactive game development.
- **Video—**Investigating and experimenting with generative AI video creation tools.
- **Chatbot Development—**Character-based chat experiences. Specialized domain experts.
- **Research and Analysis Tools—**Medical applications (drug discovery, personalized medicine). AI capability exploration. Text detection and analysis. Educational research.
- **Creative Applications—**Children's book creation. Interactive storytelling. Artistic content generation. Creative writing tools.
- **Practical Tools and Utilities—**Document generators. Translation services. Content summarizers. Professional tools (resume filters, presentation creators).

Key Takeaways

- **Try multiple genAIs**: Compare their strengths and weaknesses.
- **Master prompt engineering**: Craft effective prompts for optimal results. Build a catalog of your best prompts.
- **Be cautious of hallucinations**: Always review genAI output critically.
- **Embrace the power of genAI**: Use it to enhance your Java learning experience, not to replace it.
- **Adhere to your organization's genAI policies**.

New and Updated Features

In the following sections, we discuss the key new features and updates we've made for *Java for Programmers with Generative AI, 5/e*, including:

- **Objects-Natural Case Studies.** Chapter 1 presents a friendly introduction to object technology's basic concepts and terminology. In the early chapters, you'll create and use powerful objects of preexisting Java API classes to do significant things without knowing how to write Java classes in general or how those particular classes are implemented—and you'll do this long before you create and use objects of your own custom classes in Chapter 8. We've added objects-natural case studies on class `String` (Chapter 2), class `BigInteger` for supersized integers (Chapter 3), class `BigDecimal` for the precise monetary calculations required in business applications (Chapter 4), the Date/Time API (Chapter 5), class `ArrayList` (Chapter 6) for dynamically resizable arrays, regular expressions for locating patterns in text (Chapter 7), secret-key AES cryptography (Chapter 7) and public-key RSA cryptography (Chapter 11). Throughout the rest of the book, many other case studies use objects of Java API and open-source library classes extensively.
- **Approximately 600 integrated generative AI (genAI) prompt exercises**. Most sections end with genAI prompt exercises (approximately 450) in which you'll interact with genAIs using prompt engineering. In addition, many chapters include end-of-chapter genAI prompt exercises

(approximately 150). You'll feel like you have lots of Java experts at your side 24/7 (some receiving modest pay and some free) to answer your questions about the book's Java content, help you probe more deeply into topics of interest, and even help you write and debug Java code. Better yet, they're competing among themselves and a steady flow of new entries to offer you the best results while each rapidly improves.

- **Chapter 19, Building API-Based Java Generative AI Applications**, demonstrates programmatically interacting with OpenAI's APIs for text summarization, sentiment analysis, describing images for accessibility, translating text among languages, generating Java code, named-entity recognition, transcribing speech to text, synthesizing speech from text, creating closed captions for video, image generation and much more in the chapter's 94 exercises.
- Covers the **latest Java language features, library features and programming idioms**. To keep the book up to date as new versions of Java are released, we'll place new code examples and explanations on **our blog** at **`https://deitel.com/blog`** and link them to the book's webpage at `https://deitel.com/javafp5`.
- **Streamlined contemporary treatment of object-oriented programming**, including `record` classes, `sealed` classes and interfaces, and a new case study on programming to an interface, not an implementation, focusing on composition and dependency injection.
- **Enhanced file-processing** coverage, including using the **Jackson open-source library** to input and output data in the **JSON** and **CSV** formats that are so popular in today's data-science applications.
- **Invoking popular web services** (such as **OpenWeatherMap**) with web networking capabilities provided by the **`java.net.http` package**.
- **Special feature introducing data analytics with the CSV-format Titanic disaster dataset**.
- **Special feature on cryptography**, including **secret-key AES cryptography** and **public-key RSA cryptography**. Cryptography is crucial for the rapidly growing interest in **computer privacy and security**.

- **Special feature on simple data wrangling** steps used to prepare text for training natural language processing (NLP) and genAI models. Data wrangling is a key **data science** technology.
- New Chapter 1 sections briefly discuss the **metaverse** and its related technologies—virtual reality (VR), augmented reality (AR), mixed reality, blockchain, cryptocurrency, nonfungible tokens (NFTs) and Web3—and **software development technologies**, **big data**, and **AI's intersection with computer science and data science**.
- Enhanced treatment of **JavaFX GUI, graphics, animation and video—**technologies that will be core to the emerging metaverse. The presentation includes an intro to JavaFX's new CSS transitions for simplified animation effects.
- A brief intro to **JavaFX game development** with the **open-source FXGL (FX Game Library)** in Chapter 17.
- Flexible coverage of **JShell**, Java's friendly, command-line REPL (read-evaluate-print-loop) environment for quickly exploring, discovering and experimenting with Java's language features and libraries—**it's like having Python's interactivity in Java**.
- **Concurrency**, **Parallelism** and **Multi-Core Performance** updates for key **Project Loom** technologies like **virtual threads** and the emerging **structured concurrency** and **scoped values**.
- **Enhanced database treatment** that interactively introduces **Structured Query Language (SQL)** using the popular **SQLite database management system's command-line tools**. The examples now use **SQLite databases**. We replaced the older Swing `JTable` with **JavaFX's `TableView`**, which offers a nicer look-and-feel, better performance and more customization than `JTable`.
- Many new and updated **contemporary examples**.
- We now use `RandomGenerator` (from Java 17), which provides enhanced random-number generation capabilities and is preferred for simulations. We use `SecureRandom` in security scenarios, such as our cryptography examples.
- Our **hundreds of programming tips** gleaned from our combined ten decades of programming, teaching and industry experience have been fully integrated into the

regular text flow in this edition.

- We added various application programming case studies.
- We enhanced existing case studies and added new ones focusing on AI and data science, including simulations with random-number generation, natural language processing (NLP) and artificial intelligence via generative AI and heuristic programming.

Pedagogy Features

Syntax Coloring and Syntax Highlighting

In our e-books, we syntax color the Java code, similar to how most Java integrated development environments (IDEs) and code editors syntax color code. For the print version of the book, we use the following syntax highlighting:

Click here to view code image

```
comments appear like this
keywords appear like this
constants and literal values appear like this
all other code appears in this thin black font
```

Using Fonts for Emphasis

We emphasize on-screen components in bold (e.g., the File menu) and Java program text in a `fixed-width` font (e.g., `int x = 5;`).

Objectives

The chapter Objectives sections list the chapter's goals.

Illustrations/Figures

Abundant tables, line drawings, UML diagrams, programs and program outputs are included.

Java as It's Intended to Be Used

We use the terminology from the latest Java specification document

(https://docs.oracle.com/javase/specs/) in preference to general programming terms. We also show Java as it's intended to be used with a rich collection of application programming case studies, focusing on computer science, artificial intelligence, data science and many other fields.

Keep It Topical

To "take the pulse" of modern Java, we read, browsed or watched thousands of current articles, research papers, white papers, books, documentation pieces, blog posts, forum posts, webinars and videos. We placed Google Alerts on hundreds of important Java-related, general computing, AI and data science topics.

Hundreds of Contemporary Examples and GenAI Exercises

Our code examples and genAI exercises present current topics of interest in computing.

50+ Hours of Videos Available to O'Reilly Online Paid Subscribers

In our *Java Fundamentals with Paul Deitel, 3/e,* videos:

Click here to view code image

```
https://learning.oreilly.com/course/java-fundamentals-with/
   9780135353134/
```

co-author Paul Deitel discusses the material in the Before You Begin section, and explains and provides additional insights on most of the book's programs. You can get a free 10-day subscription at https://www.oreilly.com/start-trial/.

Performance

We focus on techniques and strategies for meeting the extraordinary performance needs of today's applications.

Data Experiences

In Chapter 11, you'll work with real-world text data. You'll read and analyze the Titanic Disaster dataset,[26] which is popular for introducing the field of **data analytics**. This dataset is stored in a

CSV (comma-separated values) file, a format we introduce in Chapter 11. Also, see the "Data Science Overlaps with Computer Science" section later in this Preface.

26. "TitanicSurvival" dataset on https://vincentarelbundock.github.io/Rdatasets. Dataset authors: Frank E. Harrell, Jr., and Thomas Cason.

Secure Java Programming

It's challenging to build industrial-strength systems that stand up to attacks from viruses and other forms of "malware." Today, via the Internet, such attacks can be instantaneous and global in scope. Building security into software from the beginning of the development cycle can significantly reduce vulnerabilities. We audited our book against the Oracle Secure Coding Guidelines for Java SE

Click here to view code image

```
https://www.oracle.com/java/technologies/javase/seccodeguide.html
```

and adhere to secure coding practices appropriate for a book at this level.

Privacy

The ACM/IEEE's curricula recommendations[27] for Computer Science, Information Technology and Cybersecurity mention privacy hundreds of times. Every programmer must consider privacy issues and concerns. We discuss secret-key and public-key cryptography in Chapters 7 and 11, respectively.

27. "Curricula Recommendations." Accessed January 18, 2025. https://www.acm.org/education/curricula-recommendations.

Ethics

The ACM/IEEE's curricula recommendations[28] for Computer Science, Information Technology and Cybersecurity mention ethics more than 100 times. In several Chapter 1 and Chapter 19 generative AI exercises, you'll focus on ethics issues via Internet search and generative AI research. You'll investigate privacy and ethical issues surrounding intelligent assistants, like Amazon Alexa, Apple Siri and generative AIs' voice capabilities. We'll also examine the excitement and controversies surrounding genAIs such as OpenAI's ChatGPT,[29] Dall-E[30] and many more.

28. "Curricula Recommendations." Accessed January 18, 2025. https://

www.acm.org/education/curricula-recommendations.
29. “Introducing ChatGPT.” Accessed January 18, 2025. https://openai.com/blog/chatgpt.
30. “Dall-E 2.” Accessed January 18, 2025. https://openai.com/product/dall-e-2.

Programming Wisdom

Hundreds of programming tips are fully integrated into the regular text flow in this edition:

- **Good programming practices** call attention to techniques that help you produce clearer, more understandable and more maintainable programs.
- We point out **common programming errors** to reduce the likelihood you’ll make them.
- **Performance tips** highlight opportunities to make your programs run faster or minimize memory use.
- **Software engineering observations** highlight architectural and design issues that affect the construction of software systems, especially large-scale systems.
- **Look-and-feel observations** (Chapters 15–17) highlight GUI conventions. These observations help you design attractive, user-friendly GUIs that conform to industry norms.

Tour of the Book

The “High-Level Overview” early in this Preface shows the book’s modular architecture. We recommend that you refer to that overview as you read this section.

The early chapters establish a solid foundation in Java fundamentals. The middle and later chapters introduce intermediate and advanced Java programming topics. We discuss six programming paradigms:

- procedural programming (Chapters 2–7)—**enhanced with our objects-natural approach**,
- object-oriented programming (8–10),

- generic programming (Chapters 12 and 13),
- functional programming (Chapter 14),
- concurrent programming (Chapter 18),
- and new in this edition—**genAI-enhanced programming** (Chapter 19)

Whether you're a student getting a sense of the book you'll be using, an instructor planning your course syllabus, or a professional software developer deciding which chapters to read as you prepare for a project, this Tour of the Book will help you make the best decisions.

Part 1: Introduction

Parts 1 and 2 (Chapters 1–7) provide a friendly, example-driven treatment of traditional introductory programming topics.

Chapter 1, Intro, Test-Driving a Java Application, and Generative AI, discusses technology trends and Internet concepts. It lays the groundwork for the Java programming discussions in Chapters 2–22 and the many case-study examples and exercises.

We discuss the technologies you'll likely use as you develop software. We introduce the Java API—existing, reusable, top-quality, high-performance capabilities that help you avoid "reinventing the wheel."

We discuss the Internet, the web, the cloud, the Internet of Things (IoT), the emerging metaverse, and exciting new developments in generative AI. We briefly mention many other topics of current interest including open-source software, virtualization, simulation, web services, multi-core hardware architecture, multithreading, natural language processing, data science, robust secure programming, cryptography, GitHub, StackOverflow, forums, blockchain, NFTs (nonfungible tokens), cryptocurrencies (like Bitcoin and Ethereum), artificial general intelligence (AGI) and more, laying the groundwork for modern Java application development.

This chapter's test-drive demonstrates compiling and executing Java code with the OpenJDK version of the Java Development Kit. We consider just how big "big data" is and how quickly it's getting even bigger. The chapter closes with an introduction to artificial intelligence (AI)—a key overlap between computer science and data science. **Includes 27 genAI prompt exercises.**

Chapter 2, Intro to Java Programming, presents Java fundamentals and illustrates key language features, including input, output, primitive data types, arithmetic operators and their precedence, and decision-making. As part of our objects-natural approach, Section 2.8's **objects-natural case study** demonstrates **creating and using objects of the Java API's class `String`**—without you having to know how to develop custom classes in general or how class `String` is implemented in particular). **Includes 12 genAI prompt exercises.**

Part 2: Additional Programming Fundamentals

Chapter 3, Control Statements: Part 1, introduces Java's control statements. You'll use the `if` and `if...else` selection statements, the `while` iteration statement for counter-controlled and sentinel-controlled iteration, and the increment, decrement and assignment operators. Section 3.12's **objects-natural case study** demonstrates using the **Java API's `BigInteger` class to create supersized integers** much larger than computer hardware can natively represent. **Includes 15 genAI prompt exercises.**

Chapter 4, Control Statements: Part 2, presents additional Java control statements—`for`, `do...while`, `switch`, `break` and `continue`—and the logical operators. Section 4.11's **objects-natural case study** demonstrates **using the Java API's `BigDecimal` class for precise monetary calculations**. **Includes 13 genAI prompt exercises.**

Chapter 5, Methods, introduces custom methods. We discuss random-number generation and simulation techniques and use them in our first of several **random-number simulation case studies** throughout the book to **implement a popular casino dice game**. We introduce the Java API's random-number generation capabilities. In Chapters 7 and 11, we use the Java API's features for producing "nondeterministic" random numbers that cannot be predicted—desirable in security applications, such as cryptography. We also discuss how the method-call stack and activation records support the method call/return mechanism.

Section 5.12's **objects-natural case study** introduces features of the Java Date/Time API, which provides robust capabilities for conveniently managing date and time information, performing date and time calculations and presenting dates and times using locale-specific formatting that considers the user's spoken (or written) language, country and time zone (that is, the user's locale). We'll

build a program that inputs a user's birth date and time (or approximations if they wish), then determines the day of the week the user was born and calculates how long the user has been alive. **Includes 16 genAI prompt exercises.**

Chapter 6, Arrays and `ArrayLists`, begins our early coverage of data structures. We present built-in arrays that store lists and tables of values. You'll define and initialize arrays and access their elements. We discuss passing arrays to methods, sorting and searching arrays and manipulating multidimensional arrays. We introduce class `Arrays`, which contains methods for performing common array manipulations. Section 6.20's **objects-natural case study** discusses creating and manipulating objects of the **Java Collection Framework's `ArrayList<E>` collection class**. The Java API provides many predefined collections (that is, data structures) that store groups of related objects in memory. These classes offer efficient, proven methods that organize, store and retrieve data portably without requiring knowledge of how the data is being stored. This reduces program development time and helps create more robust applications. **Includes 28 genAI prompt exercises.**

Chapter 7, Strings, NLP and Regex: Generative AI Foundations, presents many of the `String`, `Character` and `StringBuilder` classes' features. We present an intro to natural language processing (NLP), which is at the root of powerful generative AIs like OpenAI's ChatGPT, Google's Gemini, Anthropic's Claude, Perplexity and many others. This chapter presents two **objects-natural case studies**:

- Section 7.8 introduces **regular-expression pattern matching and text replacement** using built-in features of class `String` and features from the `java.util.regex` package. After demonstrating regular expression fundamentals, we use them to perform simple data-wrangling steps to prepare text for training NLP and generative AI models.
- Section 7.9's title—pMa5tfEKwk59dTvC04Ft1IFQz9mEXnkfYXZwxk4ujGE=—looks like gibberish. This is not a mistake! This case study continues our emphasis on security and privacy by introducing **cryptography**, which is critically important in today's connected world. Every day, cryptography is used behind the scenes to ensure your Internet-based

communications and stored data are private and secure. This case study introduces **private-key cryptography** with the AES (Advanced Encryption Standard) algorithm. In Chapter 11, our final objects-natural case study presents **public-key cryptography with the enormously popular RSA algorithm**.

Includes 22 genAI prompt exercises.

Part 3: Object-Oriented Programming

Chapter 8, Real-World Modeling with Custom Classes, begins our contemporary object-oriented programming treatment. Java is extensible—each class you create becomes a new type you can use to create objects. In Chapters 8 and 9, you'll learn Java's features for crafting valuable classes and manipulating objects of those classes, starting with a case study on creating and using a simple bank account class. In the context of a **Random-Number Simulation case study**, you'll use collections of `String`s, random-number generation and simulation techniques to implement a **text-based, card-shuffling-and-dealing program**. You can rework this case study using JavaFX (Chapters 15–17) and attractive free Wikimedia Commons card images from

Click here to view code image

```
https://commons.wikimedia.org/wiki/
   Category:SVG_English_pattern_playing_cards
```

Using a `Time` class case study and several additional classes, we continue with a deeper look at building classes, controlling access to class members and creating constructors to initialize class objects. We validate data and throw exceptions to indicate that problems have occurred. We design classes with *set* and *get* methods for changing and retrieving an object's instance variable values and discuss **composition** in which a class has references to objects of other classes as members. We discuss `enum`, `static` and `final` in more depth. We also discuss how Java reclaims unused objects (a process called garbage collection) and show a special relationship among classes in the same package. We introduce **`record` classes** for creating immutable objects that conveniently store related data items then use them to introduce pattern matching with `switch` expressions. **Includes 33 genAI prompt exercises.**

Chapter 9, Real-World Modeling with Inheritance,

Polymorphism & Interfaces, focuses on the relationships among classes in an inheritance hierarchy and the powerful runtime polymorphic processing capabilities (for conveniently "programming in the general") that these relationships enable. In this chapter's polymorphism case study, you'll implement an `Employee` class hierarchy in an application that performs polymorphic payroll calculations.

We distinguish between the **inheritance *is-a* relationship** and **composition *has-a* relationship**. Inheritance tends to create tightly coupled classes, which can make it difficult to maintain a system. We introduce interfaces, which are particularly useful for assigning common functionality to possibly unrelated classes, enabling objects of these classes to be processed polymorphically. We use a `Payable` interface implemented in disparate `Employee` and `Invoice` classes to demonstrate that objects of classes implementing the same interface can respond polymorphically to that interface's method calls.

We also explain current idioms, such as "**programming to an interface, not an implementation**" and "**preferring composition to inheritance**." We reimplement the `Employee` hierarchy using **composition and dependency-injection techniques** to create loosely coupled classes, which makes systems easier to maintain. We also introduce `sealed` classes and interfaces, which enable a class's or interface's designer to control which classes can extend a superclass or implement an interface. **Includes 31 genAI prompt exercises.**

Chapter 10, Exception Handling: A Deeper Look, continues our exception-handling discussion that began in Chapter 6. Exception handling is important for building **business-critical** applications in which a failure could disrupt company operations and cause financial losses, like

- E-commerce platforms (e.g., online stores), such as Amazon, eBay, Etsy and Shopify
- Customer Relationship Management (CRM) software, such as SalesForce, Hub-Spot and Zoho

and for building **mission-critical** applications in which failure could lead to significant financial loss, injury or even loss of life, like

- air traffic control systems,
- hospital life-support systems,

- emergency response systems used to dispatch police, firefighters and emergency medical technicians (EMTs), and
- military systems.

To use a Java component, you need to know not only how that component behaves when "things go well" but also what exceptions that component "throws" when "things go poorly."

We discuss when to use exceptions and demonstrate Java's `try` statement for catching and handling exceptions. We introduce Java's `Throwable` hierarchy and checked vs. unchecked exceptions, chained exceptions, creating custom exceptions, preconditions, postconditions, assertions and catching multiple exceptions with one `catch` handler. We also discuss the **`try`-with-resources statement** for preventing "resource leaks" by automatically releasing resources like files, network connections and database connections when a `try`-with-resources statement's `try` block terminates. This version of the `try` statement is used extensively in subsequent chapters to ensure resources like files and database connections are promptly returned to the system when no longer needed. **Includes 30 genAI prompt exercises.**

Chapter 11, Files, I/O Streams, JSON Serialization & CSV Files, demonstrates using files for data persistence and is a significant update in this edition. We begin with Java's architecture for handling files programmatically and discuss the differences between text and binary files. We demonstrate using NIO classes and interfaces to retrieve information about files and directories. Then, we create and manipulate sequential text files using `Formatter` objects to write to and `Scanner` objects to read from them.

Next, we present several case studies that demonstrate transforming objects to and from popular data formats. The first introduces **JavaScript Object Notation (JSON)**—a human-and-computer-readable text format. We use the popular **Jackson open-source library** to write objects' JSON representations to a file (a process known as serialization), then read the JSON and recreate (deserialize) the objects. The second JSON case study uses Java API networking features from the **`java.net.http` package** to invoke an `OpenWeatherMap.org` web service that returns a city's weather report in JSON format then uses Jackson to process the response and display the weather report.

Next, we introduce the **CSV (comma-separated values) file format** and how to write and read CSV files using the Jackson

open-source library. CSV is popular for datasets used in **big data**, **data analytics**, and **data science**, as well as **artificial intelligence applications** like **natural language processing**, **machine learning**, and **deep learning**—all key technologies that enable genAI. We use the Jackson library to read the **Titanic disaster dataset**, which lists information about the passengers and whether they survived when the ship struck an iceberg and sank during its maiden voyage in 1912. Then, we view some of the data and use it to introduce **basic data analytics**—a core **data science** technology.

In our final **objects-natural case study**, we continue emphasizing security by enhancing our private-key cryptography example from Section 7.9, using **RSA public-key cryptography** to encrypt and decrypt the AES secret key. This enables secure transmission of the private key to an encrypted message's recipient. Such techniques are critical for securing data and protecting user privacy. **Includes 38 genAI prompt exercises.**

Part 4: Data Structures, Generic Collections, Lambdas and Streams

Chapter 12, Generic Collections, presents our broader and deeper treatment of the Java generic collections framework that began with the **generic `ArrayList` collection** in Chapter 6. The Java collections framework contains many other **pre-built generic data structures**. We discuss the interfaces that declare the capabilities of each collection type, various classes that implement these interfaces, methods that process collection objects, and iterators that traverse collections. You'll see that **the Java API provides commonly used data structures, so you do not need to create your own**—the vast majority of your data structure needs can be fulfilled by reusing these Java API capabilities.

We introduce convenience factory methods that help you create small immutable collections. After reading **Chapter 14, Functional Programming with Lambdas & Streams**, you'll be able to reimplement many of this chapter's examples more concisely and elegantly, and in a way that makes them easier to parallelize to improve performance on multi-core systems. In Chapter 18, Concurrency: Platform Threads to Virtual Threads, you'll learn how to improve multi-core performance using Java's **concurrent collections** and **parallel stream operations**. **Includes 30 genAI prompt exercises.**

Chapter 13, Generic Classes and Methods: A Deeper Look, shows how to write custom generic classes and methods similar to those presented in **Chapter 12, Generic Collections**. We discuss how generics enable the compiler to detect type mismatches at compile time—known as **compile-time type safety**. We implement a generic method and a generic class. **Includes 24 genAI prompt exercises.**

Chapter 14, Functional Programming with Lambdas & Streams, introduces functional programming. We use lambdas and streams to write certain kinds of programs faster, simpler, more concisely and hopefully with fewer bugs than previous techniques. We architected the chapter as a series of easy-to-include-or-omit sections keyed to the book's earlier sections and chapters. We integrate lambdas and streams into several key examples after Chapter 14 because their capabilities are so convenient and compelling. Many of this chapter's sections are written so they can be covered earlier in the book. We suggest students begin with Sections 14.1–14.7 after Chapter 6 and professionals begin with Sections 14.1–14.5 after Chapter 4. We demonstrate improved ways to implement tasks you programmed in earlier chapters. After reading Chapter 14, you'll be able to cleverly reimplement many examples throughout the book. In **Chapter 18, Concurrency: Platform Threads to Virtual Threads**, you'll parallelize (i.e., perform multiple operations simultaneously) a stream operation so it can take advantage of multi-core architectures to enhance performance—a key goal of lambdas and streams. In Chapter 18, you'll also learn that it's hard to create parallel tasks that operate correctly if those tasks modify a program's state (that is, its variables' values). So, the techniques you'll learn in Chapter 14 focus on immutability—not modifying the data source being processed or any other program state. **Includes 37 genAI prompt exercises.**

(Optional) **Chapter 22, Recursion and Big O**, can be covered here.

Part 5: JavaFX Graphical User Interfaces, Graphics and Multimedia

Chapter 15, JavaFX Graphical User Interfaces: Part 1, begins our multiple-chapter feature on JavaFX graphical user interfaces (GUIs), graphics, multimedia and game-playing. Chapter 15 focuses on developing GUIs, which provide a user-friendly mechanism for interacting with applications. A GUI gives an application a distinctive "look-and-feel." Providing apps with consistent, intuitive

user-interface components gives users a sense of familiarity with a new application so they can learn it faster and use it more productively. This chapter introduces JavaFX GUI basics, including layout, controls and event handlers, which we'll implement using lambdas. You'll use the free Scene Builder tool to quickly create two GUIs with simple drag-and-drop design techniques. **Includes 13 genAI prompt exercises.**

Chapter 16, JavaFX GUI: Part 2, continues our JavaFX presentation with additional GUI layouts and controls. We show how to handle mouse and `RadioButton` events, set up event handlers that respond to property changes on controls (such as the value of a `Slider`), display shapes, bind variables to controls so the controls update automatically when the variables' data changes, and customize a GUI component's appearance. You'll also use `FileChooser` and `DirectoryChooser` dialogs to conveniently select a file or directory. Chapter 18 shows how to perform long-running tasks in a manner that allows a JavaFX GUI to remain responsive. **Includes 21 genAI prompt exercises.**

Chapter 17, JavaFX Graphics and Multimedia, presents key JavaFX graphics and multimedia capabilities, including:

- Cascading Style Sheets (CSS) for customizing JavaFX nodes' appearance and text fonts—this is based on the same technology used for decades in web development to precisely and elegantly control webpage styling.
- Two-dimensional shapes (lines, rectangles, circles, ellipses, arcs, polylines, polygons and custom paths).
- Transforming JavaFX nodes in various ways, such as translating (moving), rotating and scaling (resizing).
- Displaying videos and controlling their playback (e.g., play and pause).
- `Transition` and `Timeline` animations that dynamically change nodes' property values (such as rotation, size and location) over time.
- Frame-by-frame animations with an `AnimationTimer`.
- CSS transitions for simplifying certain animation effects.
- Canvas for drawing with the mouse.
- Three-dimensional shapes (boxes, cylinders and spheres).

We also introduce **FXGL for JavaFX game development** and are planning a deeper treatment for our blog. **Includes 31 genAI prompt exercises.**

Part 6: Advanced Topics

Chapter 18, Concurrency: Platform Threads to Virtual Threads, focuses on improving application performance and responsiveness with concurrency and multi-core programming. Chapter 18 is one of the most important chapters in the book, presenting Java's features for building applications that create and manage multiple tasks. We updated the chapter to the latest Java technologies and idioms, especially those of the game-changing Project Loom.

This chapter presents several **multithreading and multi-core systems performance case studies**. In the **Profiling Sequential and Parallel Sorting Algorithms example**, we show how to use prepackaged parallel algorithms to create multithreaded programs that will run faster (often much faster) on today's multi-core computer architectures. For example, we sort 100 million values using a sequential sort then a parallel sort. We use timing operations from Java's Date/Time API to profile the performance improvement we get on today's popular multi-core systems as we employ more cores. We show that the parallel sort runs 700% faster than the sequential sort on our computer with a 12-CPU-core Apple M2 Max processor. We include a **parallel vs. sequential stream processing** example, again using the Date/Time API to show performance improvements. The `CompletableFuture` example demonstrates sequential and parallel execution of long-running calculations.

In the **Producer/Consumer Relationship with `ArrayBlockingQueue` case study**, we discuss the classic producer–consumer relationship and demonstrate how to implement it using the predefined `ArrayBlockingQueue` class. We emphasize that **concurrent programming is difficult to get right, so you should prefer the easier-to-use, less error-prone, higher-level concurrency features**.

We've added coverage of **Project Loom**, which enhances Java's threading model with simpler, lightweight **virtual threads** (finalized in Java 21). These require significantly less memory and processor overhead than Java's traditional platform threads, enabling applications to launch enormous numbers of concurrent virtual threads and helping meet the demands of today's massively parallel applications. Project Loom reduces the complexity of

writing concurrent programs, making concurrency easier to use. We also introduce emerging Project Loom features—structured concurrency and scoped values.[31] **Structured concurrency** enables programmers to treat multiple threads as a single unit of work (a scope), making it easier to manage those threads, handle errors and reclaim resources no longer needed. **Scoped values** simplify data sharing among virtual threads in the same scope, reducing the need for complex thread-synchronization techniques. **Includes 51 genAI prompt exercises.**

31. At the time of this writing, structured concurrency and scoped values were preview features undergoing rapid change. Once they stabilize, we'll present revised live-code examples on our blog at https://deitel.com/blog.

Chapter 19, Building API-Based Java Generative AI Applications, includes many cool, fully coded API-based Java generative AI applications—see the Integrated Generative AI section's "Our Chapter 19 API-Based Complete, Live-Code Java GenAI Apps" subsection earlier in this Preface for the complete list. In the preceding chapters, we focused on manually prompting genAIs via their web interfaces and asking you to study, compare and use their results. We did not show the results of our suggested genAI prompts—you observed those in action.

When genAIs were first introduced, they typically received a text prompt and generated a specific type of output, such as text or an image. Many genAIs are now multimodal, enabling them to understand text, images, audio and even video inputs and use them to generate outputs combining text, images, audio and video. For example, you can prompt genAIs with a speech audio file and instructions asking them to transcribe the speech to text, display the transcription, analyze its sentiment (positive, negative or neutral), translate it to another spoken language and synthesize speech in that other language—and even create images and videos from the generated text! These capabilities were out of reach to most developers just a few years ago.

This chapter uses the **Simple-OpenAI open-source library** to enable Java programmers to conveniently interact with OpenAI's APIs to build powerful multimodal Java generative AI applications. You'll observe that prompt engineering is also an essential aspect of programming with genAI APIs. In the chapter examples and exercises, you'll work with genAIs to build text, speech, code, image and video apps. **Includes 94 genAI prompt and API-based coding exercises.**

Chapter 20, Accessing Databases with JDBC and SQLite, discusses database management systems (DBMSs) for conveniently storing and organizing persistent data. After an overview of database concepts, you'll use the popular open-source **SQLite** database management system's command-line tools to learn **Structured Query Language (SQL)** fundamentals. You'll execute SQL queries live to perform basic CRUD (create, read, update, delete) operations on a SQLite database and immediately see their results. Next, we use **Java's JDBC APIs** to interact programmatically with SQLite databases. JDBC is portable—so the same code we show can manipulate databases in many popular open-source and proprietary database management systems. We demonstrate connecting to databases, executing queries to update databases and retrieve data from them, and displaying query results in JavaFX `TableView`s. The chapter features a JavaFX database-driven address book application demonstrating **prepared statements** for reusable, more secure, parameterized queries. You'll enhance this app with update and delete options in the exercises. Just like genAIs can write Java code for you, they can also write SQL code, as you'll do in this chapter's genAI prompt exercises. **Includes 19 genAI prompt exercises.**

Chapter 21, Java Platform Module System (JPMS), introduces modules, which provide a higher level of aggregation above packages and enable precise organization of your code. By default, modules encapsulate implementation details while allowing you to explicitly control what is exposed to client code. Modules help developers be more productive as they build, maintain and evolve large software systems. We discuss JPMS's goals then demonstrate creating custom packages and modules, declaring module dependencies, specifying which packages a module explicitly makes available to other modules, and defining the services a module offers or consumes. We also discuss how to control reflective access between modules. Reflection allows programs to inspect and manipulate classes and their members at runtime. For example, JavaFX leverages reflection to bind a program's GUI controls to the Java code that handles user interactions with the controls. **Includes 31 genAI prompt exercises.**

Chapter 22, Recursion and Big O, discusses some key classic computer science topics. First, we use factorial and Fibonacci calculations to introduce recursive methods that call themselves directly or indirectly. We develop a JavaFX application that creates a custom "feather fractal"—a geometric figure that can be

generated from a pattern repeated recursively. We present a friendly introduction to **Big *O* notation**, which indicates how hard an algorithm may have to work to solve a problem based on the number of items it must process. **Includes 11 genAI prompt exercises. [**Chapter 22 can be covered after Chapter 14.]

Appendices

Appendix A, Introduction to JShell for Interactive Java, provides optional flexible coverage of **JShell**, Java's friendly, command-line REPL (read-evaluate-print-loop) environment that enables you to quickly explore, discover and experiment with Java's language features and libraries—**it's like having Python's interactivity in Java**. Many of this appendix's sections can be covered with the book's early chapters for instructors who'd like to incorporate this capability in their courses from the start.

JShell replaces the edit-compile-execute cycle with its **read-evaluate-print-loop (REPL)**. Rather than complete programs, you write JShell commands and Java code snippets. When you enter a snippet, JShell immediately

- **reads** it,
- **evaluates** it,
- **prints** (displays) messages that help you see the effects of your code, and
- **loops** to perform this process again for the next snippet.

The appendix is **example-intensive**. This will help you master the basics of JShell quickly. You'll see how JShell and its **instant feedback** keep your attention and speed the learning and software-development processes.

You'll find JShell easy and fun to use. It will help you learn Java features faster and more deeply and will help you verify that these features work how they're supposed to. If you are an instructor, you'll appreciate how JShell encourages your students to dig in and how it leverages the learning process. If you are a professional, you'll appreciate how JShell helps you rapidly prototype key code segments and discover and experiment with new APIs.

Appendix B, Formatted Output, presents formatting features for output and for `String`s in memory. The appendix summarizes the formatting features discussed throughout the book and introduces

additional capabilities.

Appendix C, Number Systems, discusses the binary (base 2), octal (base 8), decimal (base 10) and hexadecimal (base 16) number systems programmers use.

Key Developer Sites—GitHub and StackOverflow

"The best way to prepare [to be a programmer] is to write programs, and to study great programs that other people have written. In my case, I went to the garbage cans at the Computer Science Center and fished out listings of their operating systems."[32]—William Gates (Microsoft founder)

32. William Gates, quoted in *Programmers at Work: Interviews With 19 Programmers Who Shaped the Computer Industry* by Susan Lammers. Microsoft Press, 1986, p. 83.

You're probably already working with such popular websites as StackOverflow and GitHub. If not, check them out:

- **StackOverflow** is one of the most popular programming question-and-answer sites. Many problems you might encounter have already been discussed here. It's a great place to get your code-oriented questions answered. Many of our Google searches for various, often complex, issues throughout our writing effort returned StackOverflow posts as their first results.
- **GitHub** (now owned by Microsoft[33]) is an excellent venue for finding free, open-source code to explore and incorporate into your projects—and for you to contribute your code to the open-source community if you'd like. "Open source is software with source code that anyone can inspect, modify, and enhance."[34] We encourage you to try lots of demos and view free, open-source code examples (available on sites such as GitHub) for inspiration. Over one hundred million developers worldwide use GitHub.[35] The site hosts over 518 million code repositories in hundreds of programming languages—developers made 5.2 billion contributions to those repositories in 2024.[36] GitHub includes version-control tools that help developers manage public open-source projects and private projects. **There is a massive Java open-**

source community on GitHub where developers contribute to over 250 thousand Java code repositories.[37] We encourage you to study and execute lots of developers' open-source Java code. It's a great way to learn and is a natural extension of our live-code and objects-natural teaching approaches enhanced with generative AI.[38]

33. Microsoft, "Microsoft to Acquire GitHub for $7.5 Billion," Microsoft News, June 4, 2018. Accessed January 29, 2025. https://news.microsoft.com/2018/06/04/microsoft-to-acquire-github-for-7-5-billion/.
34. "What is open source?" Accessed January 18, 2025. https://opensource.com/resources/what-open-source.
35. "Let's build from here: The complete developer platform to build, scale, and deliver secure software." Accessed January 18, 2025. https://github.com/about.
36. "Octoverse 2024: The state of open source software." Accessed January 18, 2025. https://octoverse.github.com/.
37. "Java." Accessed January 18, 2025. https://github.com/topics/java.
38. You'll need to become familiar with the variety of open-source licenses for software on GitHub.

Deitel.com and the Deitel Blog

On the book's webpage at https://deitel.com/javafp5, we provide:

- The book's reviewer testimonials.
- A link to the book's GitHub repository for downloading the example code.
- Book updates.

For more information about downloading the code examples and setting up your Java development environment, see the Before You Begin section that follows this Preface. When we create Java-related blog posts at

```
https://deitel.com/blog
```

we'll also link each to the book's webpage at https://deitel.com/

javafp5.

Software Used in *Java for Programmers with Generative AI, 5/e*

Most of the software you'll need for this book is free for download, though the code examples in Chapter 19, Building API-Based Java Generative AI Applications, use paid web services. See the **Before You Begin** section following this Preface for download links. We wrote the Java code examples using the free OpenJDK on macOS and Windows. We developed a **utility class** used by Chapter 19's examples in the **free JetBrains IntelliJ IDEA Community Edition integrated development environment (IDE)**.

Java Documentation Links

The book includes abundant citations to research papers, articles, white papers, videos, blog posts, online documentation and more that we studied while writing the manuscript. You may want to access some of these resources for additional information and insights. The latest Java versions' documentation is available at:

Click here to view code image

```
https://docs.oracle.com/en/java/javase/index.html
```

We reference the API documentation frequently so you can get more details about the Java API capabilities we use:

Click here to view code image

```
https://docs.oracle.com/en/java/javase/23/docs/api/index.html
```

As new Java versions are released every six months, you can access their API documentation by replacing 23 in the preceding link with the new version number (24, 25, etc.).

We also frequently cite the latest Java SE (Standard Edition) Specification, which is available in both HTML and PDF forms at:

Click here to view code image

```
https://docs.oracle.com/javase/specs/
```

You can keep track of future changes at:

Click here to view code image

```
https://openjdk.org/projects/jdk/
```

Getting Your Questions Answered

Popular Java and general programming online forums include

- https://stackoverflow.com
- https://www.reddit.com/r/java/
- https://dev.to/t/java

Also, vendors often provide forums for their tools and libraries. Many libraries are managed and maintained at https://github.com. Some library maintainers offer support through the Issues tab on a given library's GitHub page.

GenAIs

You can conveniently ask genAIs Java questions and get immediate responses. We recommend that you ask multiple genAIs and compare the results. Keep in mind that genAIs sometimes make mistakes and even hallucinate, so you should verify the results with additional online searches. Even though we experienced these problems many times as we developed this manuscript, working with genAIs has leveraged our ability to create effective and engaging pedagogy. We always look forward to their responses to our prompts.

Communicating with the Authors

As you read the book, if you have questions, we're easy to reach at

```
deitel@deitel.com
```

We'll respond promptly.

Join the Deitel & Associates, Inc. Social Media Communities

You can keep up to date with Deitel on the following social media platforms:

- **LinkedIn** (https://linkedin.com/company/deitel-&-associates)
- **Facebook** (https://facebook.com/deitelfan)
- **Instagram** (https://instagram.com/DeitelFan)
- **X** (@deitel or https://x.com/deitel)
- **YouTube** (https://youtube.com/DeitelTV)
- **Mastodon** (https://mastodon.social/@deitel)

College Textbook

If you're an instructor teaching college courses, you might want to consider using ***Java How to Program: An Objects-Natural Approach, 12/e***, which is the college-textbook version of ***Java for Programmers with Generative AI, 5/e***, and is available in various digital formats. For more information, see the textbook's Preface at

```
https://deitel.com/jhtp12
```

and contact your Pearson Education representative:

Click here to view code image

```
https://pearson.com/en-us/replocator
```

Acknowledgments

Thanks to Barbara Deitel for the long hours devoted to technical research on this project. We're fortunate to have worked with Pearson's dedicated team of publishing professionals. We appreciate the guidance, wisdom and energy of Laura Lewin, Executive Editor of Pearson, IT & Professional group. Laura and her team handle all of our professional books in print and digital formats. Jackleen Sougrakpam and Erin Sullivan recruited the book's professional and academic reviewers, respectively, and managed the review process. Julie Nahil managed the production of the book's suite of print and

digital versions. Ashley Santora, Kaitlyn Coddington and Jenell Forschler helped us determine which genAIs in general and which genAI outputs in particular were acceptable to our publisher, Pearson, from a permissions perspective. Chuti Prasertsith located the cover art from Pearson's approved sources and designed several alternative covers. Jessica Deitel's design sense helped us select the final cover from those alternatives.

Reviewers

We wish to acknowledge the efforts of our reviewers—Oracle Java team members, Oracle Java Champions, other industry professionals and distinguished academics. They scrutinized the text and the programs and provided countless suggestions for improving the presentation. For this fifth edition they helped us get our new generative AI and objects-natural pedagogic approaches right. Any remaining faults in the book are our own.

We appreciate the guidance of JavaFX experts Johan Vos and Carl Dea (and for previous editions, Jim Weaver, Jonathan Giles and Simon Ritter) on the JavaFX GUI, graphics and multimedia Chapters 15–17.

Fifth Edition Professional and Academic Reviewers:

- Jeanne Boyarsky (CodeRanch, Java Champion, co-author of the *OCP Oracle Certified Professional Java SE 21 Developer Study Guide*).
- Carl Dea (Lead Software Developer at a global business consulting and services company, co-author *JavaFX 9 by Example*).
- Trisha Gee (Java Champion).
- Simon Roberts (President, Dancing Cloud Services, LLC).
- José Antonio González Seco (Parliament of Andalusia, Spain).
- Ron Veen (Java Developer, Special Agent at Team Rockstars IT) and David Vlijmincx (Senior Software Developer, JPoint), co-authors of *Virtual Threads, Structured Concurrency, and Scoped Values: Explore Java's New Threading Model*, and *Cloud-Native Development and Migration to Jakarta EE*.
- Johan Vos (Co-founder and CTO, Cloud Products at Gluon, Java Champion, co-author *The Definitive Guide to Modern Java Clients with JavaFX: Cross-Platform Mobile and Cloud Development Updated for JavaFX 21 and 23*).

- Brian Canada (Professor of Computational Science, University of South Carolina Beaufort).
- Bob Myers (Computer Science Department, Florida State University).
- Emily Navarro (Continuing Lecturer, Department of Informatics, University of California, Irvine).

A Special Thank You to Brian Goetz

Brian Goetz, Oracle's Java Language Architect and co-author of *Java Concurrency in Practice*, has reviewed several editions of our Java content over the years. His insights and constructive comments greatly enhanced our understanding of the subtleties of Java concurrency and Java object-oriented programming. He patiently guided us to "get it right." It was a privilege to be his student.

A Special Thank You to Robert Field

Robert Field, Oracle's JShell Architect reviewed the JShell appendix while the tool was under development, responding to our numerous emails in which we asked questions, reported bugs we encountered and suggested improvements. Having our content scrutinized by the person responsible for JShell and seeing some of our suggestions incorporated into the product was a special professional experience for us.

Google Search

Thanks to Google, whose search engine answers our constant stream of queries, each in a fraction of a second, at any time—and at no charge. It's one of the best productivity enhancement tools we've added to our research process in the last 20 years.

Grammarly

We use the paid version of Grammarly on all our manuscripts. They describe their product as an "AI-powered writing partner."[39] Grammarly also provides powerful free tools that can be integrated into several popular web browsers, Microsoft Office 365 and Google Docs.

39. "Grammarly." Accessed January 18, 2025. https://grammarly.com.

GenAIs

Throughout the book-development process, we used the paid versions of **OpenAI's ChatGPT**, **Google's Gemini** and **Anthropic's Claude**, and the free version of **Perplexity** to research new Java topics, verify facts, check our code for the latest Java idioms and proper commenting, suggest enhancements, ensure thorough coverage and more. Much of what we learned as we used them is reflected in the approximately 600 Generative AI prompting exercises and API-based coding exercises integrated throughout the book. The major API-based coding case studies are in Chapter 19.

Oracle, Gluon and the OpenJFX Community

Thanks to Oracle Corporation, Gluon and the OpenJFX Community, which collaborate on JavaFX's evolution, releasing a new JavaFX version with each new Java SDK version. Also, thanks to Gluon for their friendly and convenient drag-and-drop Scene Builder GUI development tool.

Welcome to the exciting world of Java programming enhanced with generative AI. We've enjoyed writing many editions of our professional and academic Java content over the last 30 years. We hope you have an informative, challenging and entertaining learning experience using ***Java for Programmers with Generative AI, 5/e***, and enjoy this look at modern Java software development.

As you read the book and work through the code examples and generative AI exercises, we'd appreciate your comments, criticisms, corrections and suggestions for improvement. Please send all correspondence, including questions, to

```
deitel@deitel.com
```

We'll respond promptly. Our best wishes for your success,

Paul Deitel
Harvey Deitel

About the Authors

Paul J. Deitel, CEO and Chief Technical Officer of Deitel & Associates, Inc., is a graduate of MIT with over 44 years of experience in computing. He holds the Java Certified Programmer and Java Certified Developer designations and is an Oracle Java Champion. He is one of the world's most experienced programming-

languages trainers, having taught hundreds of professional courses to software developers since 1992. His video and live-training courses on platforms like O'Reilly Online Learning have garnered millions of contact minutes, with his Java Fundamentals, Python Fundamentals and C# Fundamentals video courses each ranking #1 for several years among thousands of video products. He has delivered programming courses to academic, industry, government and military clients of Deitel & Associates, Inc. internationally, including UCLA, SLB (formerly Schlumberger), IBM, Siemens, Sun Microsystems (now Oracle), Dell, Fidelity, NASA at the Kennedy Space Center, the National Severe Storm Laboratory, White Sands Missile Range, Rogue Wave Software, Boeing, Cisco, Puma, iRobot and many more. He and his co-author, Dr. Harvey M. Deitel, are among the world's best-selling programming-language textbook/professional book/video/interactive multimedia authors.

Dr. Harvey M. Deitel, Chairman and Chief Strategy Officer of Deitel & Associates, Inc., is recognized as a thought leader and innovator in programming languages education. He has over 64 years of experience in computing. Dr. Deitel earned B.S. and M.S. degrees in Electrical Engineering from MIT and a Ph.D. in Mathematics from Boston University—he studied computing in each of these programs before they spun off Computer Science programs. He has extensive college teaching experience, including earning tenure and serving as the Chairman of the Computer Science Department at Boston College before founding Deitel & Associates in 1991 with his son, Paul. The Deitels' publications have earned international recognition, with more than 100 translations published in Japanese, German, Russian, Spanish, French, Polish, Italian, Simplified Chinese, Traditional Chinese, Korean, Portuguese, Greek, Urdu and Turkish. Dr. Deitel has delivered hundreds of programming courses to academic, corporate, government and military clients.

About Deitel® & Associates, Inc.

Deitel & Associates, Inc., founded by Paul Deitel and Harvey Deitel, is an internationally recognized authoring and corporate-training organization specializing in computer programming languages, object technology, mobile app development, Internet and web software technology and generative AI. The company's training clients include some of the world's largest companies, government agencies, branches of the military, and academic institutions. The

company offers instructor-led training courses delivered live at client sites worldwide and virtually worldwide for Pearson.

Through its five-decade publ5bwishing partnership with Pearson, Deitel & Associates, Inc. publishes leading-edge computer programming college textbooks and professional books in print and digital formats, video courses, O'Reilly Online Learning live training courses and Pearson + eText and Revel interactive multimedia college courses.

To contact Deitel & Associates, Inc. and the authors or to request a proposal for virtual or on-site, instructor-led training worldwide, write to

```
deitel@deitel.com
```

To learn more about Deitel corporate training, visit

```
https://deitel.com/training
```

Individuals wishing to purchase Deitel print and digital publications online can do so at

```
https://amazon.com
https://barnesandnoble.com/
```

For convenient access to updates and corrections as they become available, register your copy of *Java for Programmers with Generative AI, 5/e*, on Pearson's InformIT site. To start the registration process, go to

```
https://informit.com/register
```

and log in or create an account. Enter the product ISBN (`9780137574735`) and click Submit.

Deitel e-books are available in various formats from

Click here to view code image

```
https://amazon.com/              https://vitalsource.com/
https://barnesandnoble.com/      https://redshelf.com/
https://informit.com/
```

To register for a free 10-day trial to O'Reilly Online Learning, visit

Click here to view code image

```
https://learning.oreilly.com/register
```

Before You Begin

Please read this section to understand our conventions and set up your computer to compile and run our example programs. If there are changes to the instructions presented here, we'll post updates on the book's webpage at

```
https://deitel.com/javafp5
```

Font and Naming Conventions

We use fonts to distinguish application elements and Java code elements from regular text:

- We use a **bold font** for on-screen application elements, such as "the **File** menu."
- We use a `fixed-width font` for commands and Java code elements, as in

  Click here to view code image

  ```
  System.out.println("Welcome to Java!");
  ```

Getting the Code Examples

Java for Programmers with Generative AI, 5/e's examples are available for download from our GitHub repository at

Click here to view code image

```
https://github.com/pdeitel/JavaForProgrammers5e
```

If you're not familiar with GitHub, press (or select) the green "< > **Code**" button, then select **Download ZIP** from the drop-down menu that appears. This will save

```
JavaForProgrammers5e-main.zip
```

in your user account's `Downloads` folder. Next, extract the ZIP file's contents:

- On macOS, double-click the file to extract its contents, then navigate into the `JavaForProgrammers5e-main` folder and drag the `examples` folder to your `Documents` folder.
- On Windows, double-click the file, navigate into the `JavaForProgrammers5e-main` folder, then drag the `examples` folder to your `Documents` folder.

Various instructions and code examples throughout the book assume the examples are in a folder named `examples` in your user account's `Documents` folder, structured as follows:

```
Documents
   examples
      ch01
      ch02
      ch03
      ...
```

If you place the `examples` folder elsewhere or rename it, you must update our instructions and code accordingly.

Java Development Kit (JDK)

We developed this book's examples using the Java 23 version of the Java Development Kit (JDK), which contains the command `javac` for compiling Java programs, the command `java` for running Java programs, and more. We discuss compiling and running Java programs in Chapter 1's "Test-Driving the JDK" section.

We used the free, open-source OpenJDK version of the JDK from:

Click here to view code image

```
https://adoptium.net/temurin/releases/
```

This site—managed by the Eclipse Foundation[1]—provides downloads for many operating systems and hardware platforms.

Below, we discuss getting the OpenJDK for Windows and macOS. You may also use newer JDK versions with this book.

1. "About the Eclipse Foundation." Accessed February 12, 2025. https://eclipse.org/org/.

Windows Users

Go to

Click here to view code image

```
https://adoptium.net/temurin/releases/
```

then select the following options:

- Under **Operating System**, select **Windows**.
- Under **Architecture**, select **x64**.
- Under **Package Type**, select **JDK**.
- Under **Version**, select **23** (or higher to use a newer version).

Next, press (or select) the `.msi` installer button to download the OpenJDK installer. When the download completes, double-click the installer file in your user account's `Downloads` folder. The installer's filename will be similar to:

Click here to view code image

```
OpenJDK23U-jdk_x64_windows_hotspot_23.0.2_7.msi
```

In the installed dialog:

- Press (or select) **Next** to move to the **End-User License Agreement** step. You must accept the license agreement to install the software. Once you've read the agreement, select "I accept the terms in the License Agreement," then press (or select) **Next** to move to the **Installation Scope** step.
- Choose whether to install the JDK just for you or for anyone using your computer, then press (or select) **Next** to move to the **Custom Setup** step.
- By default, the installer selects the first three options (**Modify PATH variable**, **Associate .jar** and **Set or override**

JAVA_HOME variable). We recommend keeping the default settings and installation location. Press (or select) **Next**, then click **Install**.

- In the **User Account Control** dialog, press (or select) **Yes** to allow the installer to make changes to your system and install the JDK.
- When the installer completes, press (or select) **Finish**.

To ensure that Java is installed correctly, enter **Terminal** in your Windows taskbar's **Search** field then select **Terminal** in the search results. This opens a **Windows PowerShell** window. In this window, enter the command

```
java --version
```

and press (or select) *Enter* or *Return* to execute it. The command's output should be similar to the following:

Click here to view code image

```
openjdk 23.0.2 2025-01-21
OpenJDK Runtime Environment Temurin-23.0.2+7 (build 23.0.2+7)
OpenJDK 64-Bit Server VM Temurin-23.0.2+7 (build 23.0.2+7, mixed mode, sharing)
```

MacOS Users

Go to

Click here to view code image

```
https://adoptium.net/temurin/releases/
```

then choose the following options:

- Under **Operating System**, select **macOS**.
- Under **Architecture**, for Macs with Apple silicon (M-series processors), select **aarch64**; otherwise, select **x64** for Intel-processor Macs.
- Under **Package Type**, select **JDK**.
- Under **Version**, select **23** (or higher to use a newer version).

Next, press (or select) the `.pkg` installer button to download the OpenJDK installer. When the download completes, double-click the

installer file in your user account's `Downloads` folder. The installer's filename for M-processor Macs will be similar to:

Click here to view code image

```
OpenJDK23U-jdk_aarch64_mac_hotspot_23.0.2_7.pkg
```

and for Intel-processor Macs will be similar to:

Click here to view code image

```
OpenJDK23U-jdk_x64_mac_hotspot_23.0.2_7.pkg
```

In the installer dialog:

- Press (or select) **Continue** to move to the **Software License Agreement** step. You must accept the license agreement to install the software. Once you've read the agreement, press (or select) **Continue**, then press (or select) **Agree** to accept the license terms and proceed to the **Standard Install on "Macintosh HD"** step.
- Press (or select) **Install** to perform the installation. If you are asked for your system password, enter it and press (or select) **Install Software**.
- When the installation is complete, press (or select) **Close** in the dialog.

To ensure that Java is installed correctly, in your **Finder**, select the **Go** menu's **Utilities** option, then open **Terminal.app**. In the **Terminal** window, type the command

```
java --version
```

and press (or select) *Enter* or *Return* to execute it. The command's output should be similar to the following:

Click here to view code image

```
openjdk 23.0.2 2025-01-21
OpenJDK Runtime Environment Temurin-23.0.2+7 (build 23.0.2+7)
OpenJDK 64-Bit Server VM Temurin-23.0.2+7 (build 23.0.2+7, mixed mode, sharing)
```

Java Release Cadence

Java follows a six-month release cadence that began with Java 9 in

2017. New versions are released every March and September, enabling faster delivery of new language and library features. Every two years, a release is designated as a Long-Term Support (LTS) version. These are geared toward enterprises requiring stability and extended support. Non-LTS releases have shorter support cycles, encouraging early adoption of innovations. The latest LTS version is Java 21. The next will be Java 25 (September 2025).

Java Integrated Development Environments (IDEs)

There are many Java integrated development environments (IDEs)—software applications that combine essential tools for software development, such as a code editor, compiler, debugger and other tools to help you build applications. The steps for using each differ, so we use the JDK's command-line tools for most examples. In Chapter 1, we provide a JDK test-drive. On the book's webpage

```
https://deitel.com/javafp5
```

we provide the following IDE test-drives:

- JetBrains' IntelliJ IDEA Community Edition,
- the Eclipse Foundation's Eclipse IDE for Java Developers, and
- Microsoft's Visual Studio Code cross-platform code editor—this tool has plugins for Java development.

These IDEs have the most significant market shares in the Java development community. Each is available for Windows, macOS and Linux. **Before installing any of the tools we discuss below, install the OpenJDK as described in the previous section**.

IntelliJ IDEA Community Edition

The recommended way to install the IntelliJ IDEA Community Edition IDE is via the Jet-Brains **Toolbox** app, which you can download from:

Click here to view code image

```
https://jetbrains.com/toolbox-app/
```

The site will recognize your operating system and provide a **Download** button for the appropriate installer. Press (or select) **Download**. When the download completes, double-click the installer file in your `Downloads` folder and follow the on-screen prompts to install the **Toolbox** app. Next, press (or select) the **Toolbox** app icon to open its window:

- macOS users will find the **Toolbox** app icon at the top-right of the screen.
- Windows users will find the **Toolbox** app icon in the task tray at the bottom-right of the screen. You may need to press (or select) the small upward-pointing arrow to see all the task tray icons.

In the **Toolbox** window, scroll to **IntelliJ IDEA Community Edition** and press (or select) its **Install** button.

Eclipse IDE for Java Developers

The Eclipse Foundation provides step-by-step Eclipse IDE installation instructions at

Click here to view code image

```
https://eclipse.org/downloads/packages/installer
```

In Step 3, choose the **Eclipse IDE for Java Developers**, then follow the remaining instructions to complete the installation.

Visual Studio Code

Microsoft provides step-by-step Visual Studio Code download and install instructions for Windows users at:

Click here to view code image

```
https://code.visualstudio.com/docs/setup/windows
```

and for macOS users at:

Click here to view code image

```
https://code.visualstudio.com/docs/setup/mac
```

After installing Visual Studio Code, follow Microsoft's step-by-step instructions for getting started with Java at:

Click here to view code image

```
https://code.visualstudio.com/docs/java/java-tutorial
```

JavaFX Software Development Kit (SDK)

In Chapters 15–17, you'll build graphical user interface (GUI), graphics and multimedia apps using JavaFX, which must be installed separately from the JDK. JavaFX is developed in parallel with the JDK by a consortium of companies and individuals, including Oracle, which oversees Java's development. A new JavaFX version is released in parallel with each new Java version.

Download the JavaFX SDK

To work with JavaFX, you must first install the JavaFX Software Development Kit (SDK), which is available from:

Click here to view code image

```
https://gluonhq.com/products/javafx/
```

In the **Downloads** section, choose the following options:

- Under **JavaFX version**, select the version that matches your JDK version (JavaFX 23.0.2 at the time of this writing).
- Under **Operating System**, select your operating system.
- Under **Architecture**, Apple silicon (M-series processor) Mac users should select **aarch64**, and Intel-processor Mac users and Windows users should select **x64**.
- Under **Package Type**, select **SDK**.

Next, press (or select) the **Download** button to download the SDK for your platform. This places a ZIP file in your `Downloads` folder with a name similar to

Click here to view code image

```
openjfx-23.0.2_OS-architecture_bin-sdk.zip
```

where *OS* is your operating system and *architecture* is the architecture you selected.

Extract the JavaFX SDK from the ZIP File

Next, extract the ZIP file's contents:

- On macOS, double-click the file to extract its contents. This creates a folder in your `Downloads` folder named `javafx-sdk-23.0.2` (the version number will depend on the precise version you downloaded).
- On Windows, double-click the file, then drag the `javafx-sdk-23.0.2` folder to your `Downloads` folder (the version number will depend on the precise version you downloaded).

To complete the installation, you must set the `PATH_TO_FX` environment variable, as discussed below for Windows and macOS.

Configuring the `PATH_TO_FX` Environment Variable on Windows

Perform the following steps:

- Enter `environment` in the taskbar's **Search** field.
- In the search results, select **Edit the system environment variables**.
- In the **System Properties** dialog, press (or select) the **Environment Variables** button.
- Under **User variables**, click **New...**.
- Enter the **Variable name** `PATH_TO_FX`. Then, for the **Variable value**, enter the following, replacing *User* with your user account's folder name and, if necessary, replacing `23.0.2` with your JavaFX version:

  Click here to view code image

  ```
  C:\Users\User\Downloads\javafx-sdk-23.0.2\lib
  ```

- Press (or select) **OK** to save the environment variable.
- Press (or select) **OK** in the **System Properties** dialog to close it.

If you have any open Command Prompt or PowerShell windows, you'll need to close and reopen them for the environment variable to take effect.

Configuring the `PATH_TO_FX` Environment Variable on macOS

Perform the following steps:

- In your **Finder**, select the **Go** menu's **Utilities** option, then open **Terminal.app**.
- Next, determine your shell executing the command

  ```
  echo $SHELL
  ```

- If Step 2 displayed `"/bin/zsh"`, execute

  ```
  open -e ~/.zshrc
  ```

  Otherwise, execute

  ```
  open -e ~/.bash_profile
  ```

- Add the following line to the end of the file, replacing *User* with your user account's folder name and, if necessary, replacing `23.0.2` with the JavaFX version you downloaded:

  Click here to view code image

  ```
  export PATH_TO_FX="/Users/User/Downloads/javafx-sdk-23.0.2/lib"
  ```

  then save and close the file.

Scene Builder

In our JavaFX examples, you'll use the free Scene Builder tool to create graphical user interfaces (GUIs) using drag-and-drop techniques. To download the Scene Builder installer, press (or select) the **Download Now** button at:

Click here to view code image

```
https://gluonhq.com/labs/scene-builder/
```

Next, press (or select) the **Download** button for your platform. When the download completes, double-click the installer, then follow the on-screen prompts to install Scene Builder.

Generative AIs

As mentioned in the Preface section "Integrated Generative AI," you'll leverage generative AI (genAI) as you work your way through this book. At Deitel & Associates, Inc., we use many genAIs, including the following four genAI chatbots:

- OpenAI ChatGPT (`https://chatgpt.com`)
- Google Gemini (`https://gemini.google.com/app`)
- Anthropic Claude (`https://claude.ai`)
- Perplexity (`https://perplexity.ai`)

Registering for Accounts

You can use ChatGPT and Perplexity for free without registering. Google requires you to sign into your Google account to use Gemini's free version. For Claude, you must provide an email address to use the free version, then they'll send you a login link. ChatGPT and Perplexity will remember your past chats only if you register for a free account.

Paid vs. Free Tiers

We use these chatbots' paid versions, but each offers a powerful free tier. However, the free versions are more limited. They generally

- allow fewer queries in a given time period,
- allow shorter queries,
- allow fewer (or disallow) file uploads,
- have slower response times and
- limit or prevent access to their most up-to-date and powerful capabilities.

Using the GenAI Chatbots

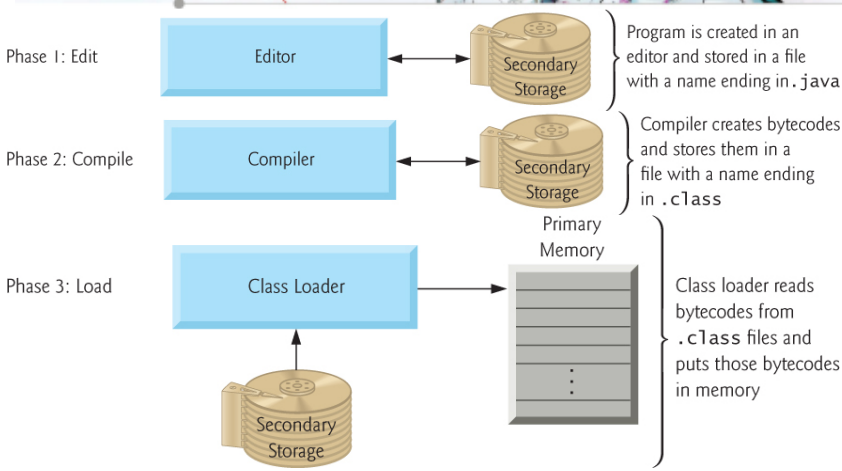
We'll suggest hundreds of genAI prompts throughout this book, so you'll use genAI chat-bots extensively. When you visit each chatbot, its webpage will provide a box in which you can enter a prompt and submit it to the chatbot. ChatGPT and Gemini also allow you to

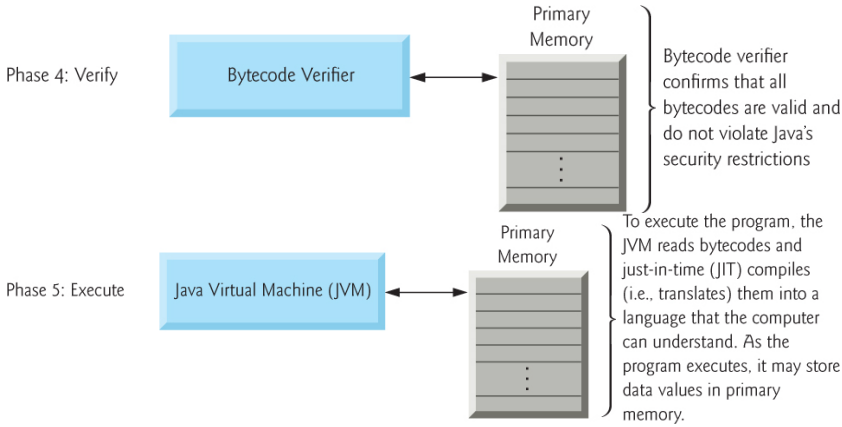
speak a prompt. To get a sense of each chatbot's capabilities, submit prompts like:

Click here to view code image

> *What are your capabilities?*
>
> *What are the differences between your free and paid tiers?*

When you submit a prompt, the chatbot will formulate and present its response. And as a reminder: **Always verify the information genAI chatbots provide.**


Phase 1: Edit
Editor
Secondary Storage
Program is created in an editor and stored in a file with a name ending in .java
Phase 2: Compile
Compiler
Secondary Storage
Compiler creates bytecodes and stores them in a file with a name ending in .class
Primary Memory
Phase 3: Load
Class Loader
Secondary Storage
Class loader reads bytecodes from .class files and puts those bytecodes in memory



Intro, Test-Driving a Java Application, and Generative AI

Objectives

In this chapter, you'll:

- Understand the strengths of Java.
- Be introduced to the Java Application Programming Interface (Java API) of reusable components.
- Learn a typical Java program-development environment.

- Test-drive the Java Development Kit (JDK).
- Review Internet, web, cloud and IoT basics.
- Review object-technology concepts that will be used in the objects-natural case studies (Chapters 2–7 and 11) and discussed in detail in the object-oriented programming Chapters 8–9 then book-wide.
- Understand how concurrent programming helps maximize performance on multi-core processors.
- Be introduced to big data and data science technologies.
- Learn about exciting recent developments in computing, including the Metaverse, artificial intelligence and generative AI.

Outline

1.1 Introduction

1.2 Hardware

1.3 Java

1.4 A Brief Review of Object Orientation

1.5 Java Application Programming Interface (API) and Open-Source Libraries

1.6 Typical Java Program Development Environment

1.7 Test-Driving a Java Program with the Java Development Kit (JDK)

1.7.1 Compiling and Running the Program with `javac` and `java`

1.7.2 Running the Program's Source-Code File Directly with the `java` Command

1.8 Internet, World Wide Web, the Cloud and IoT

1.8.1 The Internet: A Network of Networks

1.8.2 The World Wide Web: Making the Internet User-Friendly

1.8.3 The Cloud

1.8.4 The Internet of Things (IoT)

1.8.5 Edge Computing

1.8.6 Mashups

1.9 Metaverse

1.9.1 Virtual Reality (VR)
1.9.2 Augmented Reality (AR)
1.9.3 Mixed Reality
1.9.4 Blockchain
1.9.5 Bitcoin and Cryptocurrency
1.9.6 Ethereum
1.9.7 Non-Fungible Tokens (NFTs)
1.9.8 Web3
1.10 Software Development Technologies
1.11 Data Analytics and Data Science
1.12 How Big Is Big Data?
1.12.1 Big-Data Analytics
1.12.2 Data Science and Big Data Are Making a Difference: Use Cases
1.13 AI—at the Intersection of Computer Science and Data Science
1.13.1 Artificial Intelligence (AI)
1.13.2 Artificial General Intelligence (AGI)
1.13.3 Artificial Intelligence Milestones
1.13.4 Machine Learning
1.13.5 Deep Learning
1.13.6 Reinforcement Learning
1.14 Generative AI
1.15 Wrap-Up
Special Section: Generative AI Exercises

1.1 Introduction

Welcome to Java—one of the world's most popular and senior computer programming languages.[1,2,3] This book provides an intensive, hands-on Java programming experience.

1. "TIOBE Index for February 2025." Accessed February 18, 2025. https://tiobe.com/tiobe-index/.
2. "The Top Programming Languages 2024." Accessed February 18, 2025. https://spectrum.ieee.org/top-programming-languages-2024.
3. "PYPL PopularitY of Programming Language." Accessed February 18,

2025. https://pypl.github.io/PYPL.html.

The chapter introduces terminology and concepts that lay the groundwork for the Java programming capabilities you'll learn, beginning in Chapter 2. Our intro to object orientation presents the terminology we use as we present Java object-oriented programming throughout this book. We introduce Java and discuss the Java Application Programming Interface (Java API) and various Java "open-source" libraries that help you avoid "reinventing the wheel." We present a typical Java program-development environment in the context of the Java Development Kit (JDK), using it to test-drive a Java program (Section 1.7) to prepare you to run the book's examples. You can also run our programs using one of the many Java integrated development environments (IDEs). We provide test-drives for several popular IDEs at

```
https://deitel.com/javafp5
```

We review the Internet, the web, the cloud and the Internet of Things (IoT) basics. We introduce the Metaverse and its related technologies, including blockchain, cryptocurrencies, NFTs, Web3, augmented reality, virtual reality and mixed reality. We briefly introduce several contemporary software-development technologies. Finally, we introduce big data, data science and artificial intelligence (AI) technologies—machine learning, deep learning and exciting generative AIs like OpenAI's ChatGPT, Google's Gemini and Anthropic's Claude. You'll use these and other genAIs throughout this book to leverage your Java learning experience and do some really cool stuff, beginning with this chapter's Special Section: Generative AI Exercises!

This Book's Architecture

Before you "dig in," we recommend getting a "40,000-foot" view of the book's architecture to understand where you're headed as you prepare to learn and use the powerful Java programming language. To do so, review the following items:

- In the Preface, the **High-Level Overview** presents the book's modular architecture in a convenient condensed format. The Preface also contains a **list of the book's key features**.
- At https://deitel.com/javafp5, we provide **testimonial**

comments from Java subject-matter experts and university professors who reviewed the prepublication manuscript.

- The Preface presents the "soul of the book" and our approach to teaching modern Java programming. We introduce the early chapters' "objects-natural approach," in which you'll use small numbers of simple Java statements to make objects of powerful preexisting classes perform significant tasks—long before you create your own custom Java classes in Chapter 8. Be sure to read the Preface's Tour of the Book, which points out the key features of each chapter and enumerates the book's more substantial case studies and generative AI exercises. You might also want to refer to the Preface's High-Level Overview as you read the Tour.

Java Is Widely Used

There are billions of personal computers in use and an even larger number of mobile devices with computers at their core. Various estimates indicate that 7–9 million Java developers worldwide are writing Java code that runs on the 60 billion installed Java Virtual Machines (JVMs).[4] In addition, the explosive growth of mobile devices is creating significant opportunities for programming mobile apps. The following table shows some device types that Java runs on:

4. Oracle. "Java." Accessed November 26, 2024. https://www.oracle.com/java/.

Devices
Desktop computers
Cable boxes
Copiers
Automobile navigation systems
Blu-ray Disc players
Optical sensor systems
Smartphones
Point-of-sale terminals
Lights
Robotic controllers
Medical devices
Mobile phones
Set-top boxes
Automobile diagnostic systems

Java Standard Edition (Java SE)

This fifth edition of *Java for Programmers* is based on the **Java Standard Edition**, which contains the capabilities needed to run desktop and server applications.[5] Java supports several programming paradigms that we discuss in this book:

- procedural programming (Chapters 2–7),
- object-oriented programming (Chapters 8–10),
- generic programming (Chapters 12 and 13),
- functional programming (Chapter 14),
- concurrent programming (Chapter 18) and
- generative-AI-assisted programming (Chapter 19).

5. Oracle. "Java SE at a Glance." November 26, 2024. https://www.oracle.com/java/technologies/java-se-glance.html.

Jakarta EE

Java is used in such a broad spectrum of applications that it has two other editions. **Jakarta EE**—formerly called **Java Enterprise Edition (Java EE)**—is geared toward developing the kinds of large-scale, distributed networking and web-based applications that run major organizations.

Java Micro Edition

The **Java Micro Edition (Java ME)**—a subset of Java SE—is geared toward applications for resource-constrained embedded devices, such as smartwatches, medical devices, television set-top boxes, smart meters (e.g., for monitoring energy usage) and more.

1.2 Hardware

Computing costs are dropping dramatically due to rapid developments in hardware and software technologies. Computers that might have filled large rooms and cost millions of dollars decades ago are now inscribed on silicon computer chips smaller than a fingernail, many costing perhaps a few dollars each. Ironically, silicon is one of the most abundant materials on Earth—it's an ingredient in ordinary sand. Silicon-chip technology has made computing so economical that computers and computerized

devices have become commodities.

Moore's Law, Multi-Core Processors and Concurrent Programming

Many of today's personal computers can perform trillions of calculations in one second[6,7]—far more than a human can perform in a lifetime. You probably expect to pay at least a little more every year for most products and services. The opposite has been the case in the computer and communications fields, especially with regard to the hardware supporting these technologies. Over the years, hardware costs have fallen rapidly.

6. Tobias Mann, "Intel's Lunar Lake Mobile Chips Boast 120 TOPS of AI Performance," The Register, September 3, 2024. Accessed February 18, 2025. https://www.theregister.com/2024/09/03/intels_lunar_lake_mobile_chips/.
7. Tobias Mann, "Apple's M4 AI Chip: A Leap in Performance," The Register, October 31, 2024. Accessed February 18, 2025. https://www.theregister.com/2024/10/31/apple_m4_ai_chip/.

For decades, computer processing power approximately doubled inexpensively every couple of years. This remarkable trend is called **Moore's Law**—named for Gordon Moore, co-founder of Intel and the person who identified the trend in the 1960s. Intel is a leading manufacturer of processors for today's computers and embedded systems, such as smart home appliances, home security systems, robots, intelligent traffic intersections and more. Moore's Law and related observations especially apply to:

- the amount of memory that computers have for programs and data,
- the amount of secondary storage they have to hold programs and data, and
- their processor speeds—the speeds at which computers execute programs to do their work.

Key executives at computer-processor companies NVIDIA and ARM have indicated that Moore's Law no longer applies. However, Intel's CEO says it is "alive and well"—they are working on manufacturing advances that will enable more transistors on each chip.[8] Computer processing power continues to increase but relies

on new processor designs, such as multi-core processors.

8. Christian Hetzner, “Intel CEO Pat Gelsinger Dismisses Nvidia’s AI Chip Dominance: ‘Moore’s Law Is Alive and Well,’” Observer, June 2024. Accessed February 18, 2025. https://observer.com/2024/06/intel-ceo-pat-gelsinger-nvidia-ai-chip-rivalry/.

Multi-Core Processors and Performance

Most computers today have multi-core processors that economically implement multiple processors on a single integrated circuit chip. Intel has some processors with up to 128 cores.[9] Our primary testing computer uses Apple’s M2 Max processor, which has a 12-core CPU, a 28-core GPU and a 16-core “neural engine” for AI tasks.[10] Apple’s most recent M4 Max has up to a 16-core CPU, up to a 40-core GPU and a 16-core “neural engine” for AI tasks.[11] AMD has processors with up to 192 cores.[12] The number of cores will continue to grow. Today’s most powerful GPUs used in high-end gaming computers and artificial intelligence applications often have thousands of cores in their GPUs. For example, the NVIDIA RTX 5090 has 21,760 cores![13]

9. “Intel® Xeon® 6 Processors with Performance-Cores (P-Cores),” Intel. Accessed February 18, 2025. https://www.intel.com/content/www/us/en/products/details/processors/xeon/xeon6-pcores.html.

10. “Apple unveils M2 Pro and M2 Max: next-generation chips for next-level workflows,” January 23, 2023. Accessed February 18, 2025. https://www.apple.com/newsroom/2023/01/apple-unveils-m2-pro-and-m2-max-next-generation-chips-for-next-level-workflows/.

11. “Apple introduces M4 Pro and M4 Max,” October 30, 2024. Accessed February 18, 2025. https://www.apple.com/newsroom/2024/10/apple-introduces-m4-pro-and-m4-max/.

12. “EPYC™ 9005 Series Processors,” AMD. Accessed February 18, 2025. https://www.amd.com/en/products/processors/server/epyc/9005-series.html.

13. Dave James, “Nvidia GeForce RTX 5090,” PC Gamer, January 30, 2025. Accessed February 18, 2025. https://www.pcgamer.com/nvidia-geforce-rtx-5090/.

In multi-core systems, the hardware can put multiple processors to work truly simultaneously on different parts of your task, enabling your program to complete faster. **Taking full advantage of multi-core architecture requires writing multithreaded applications (Chapter 18).** When a program’s task is split into separate threads, a multi-core system can run those threads in parallel when a sufficient number of cores is available.

Interest in multithreading is rising quickly with the proliferation of multi-core systems. Java has provided multithreading since its inception and has added higher-level capabilities over the years to simplify multithreaded application development. **Chapter 18, Concurrency: Platform Threads to Virtual Threads**, discusses creating and managing multi-threaded Java applications. It includes a detailed explanation with extensive code examples of the latest simplification technologies to emerge from Project Loom—virtual threads, structured concurrency and scoped values.

Computing Power Over the Years

Data is getting more massive, and so is the computing power needed to process it. Today's processor performance is often measured in terms of **FLOPS (floating-point operations per second)**. In the 1990s, the fastest **supercomputer** speeds were measured in gigaflops (10^9 FLOPS). By the late 1990s, Intel produced the first teraflop (10^{12} FLOPS) supercomputers. In 2008, IBM released the first petaflop (10^{15} FLOPS) supercomputer. Today's fastest supercomputers can perform over one million trillion (a quintillion) instructions per second! As of November 2024, the USA's El Capitan supercomputer (from Lawrence Livermore National Laboratory) is the world's fastest[14]—1.742 quintillion calculations per second (1.742 exaflops)![15] For perspective, it can perform in one second approximately 215 million calculations per person for everyone on the planet![16]

14. "Top 500." Wikipedia. Wikimedia Foundation. Accessed February 18, 2025. https://en.wikipedia.org/wiki/TOP500.

15. "Flops." Wikipedia. Wikimedia Foundation. Accessed February 18, 2025. https://en.wikipedia.org/wiki/FLOPS.

16. For perspective on how far computing performance has come, consider this: In his early computing days in the 1960s, one of the authors, Harvey Deitel, used the Digital Equipment Corporation PDP-1 (https://en.wikipedia.org/wiki/PDP-1), which was capable of performing only 93,458 operations per second, and the IBM 1401 (https://www.ibm-1401.info/1401GuidePosterV9.html), which performed only 86,957 operations per second.

The **quantum computers** now under development theoretically could operate at 18 quintillion (18,000,000,000,000,000,000) times the speed of today's computers![17] This number is so extraordinary that in one second, a quantum computer could do staggeringly more calculations than the total that have been performed by all computing devices of any kind since the beginning of time. In

December 2024, Google announced its new Willow quantum computing chip, which took five minutes to perform a task that would take a conventional computer 10 septillion (10,000,000,000,000,000,000,000,000) years to complete.[18] This almost unimaginable computing power could wreak havoc with blockchain-based cryptocurrencies like Bitcoin. Engineers are already rethinking blockchain[19] and working on post-quantum cryptography to prepare for such massive increases in computing power.[20]

17. Norbert Biedrzycki, "Only God can count that fast — the world of quantum computing." May 12, 2017. Accessed February 18, 2025. https://medium.com/@n.biedrzycki/only-god-can-count-that-fast-the-world-of-quantum-computing-406a0a91fcf4.
18. Robert Booth "Google Unveils 'Mindboggling' Quantum Computing Chip," The Guardian, December 9, 2024. Accessed February 18, 2025. https://www.theguardian.com/technology/2024/dec/09/google-unveils-mindboggling-quantum-computing-chip.
19. "Blockchain." Wikipedia. Wikimedia Foundation. Accessed February 18, 2025. https://en.wikipedia.org/wiki/Blockchain.
20. Dr. Reji Thomas, "Quantum-Secure Blockchain: Preparing For The Era Of Quantum Computing," November 26, 2024. Accessed February 18, 2025. https://www.forbes.com/councils/forbestechcouncil/2024/11/26/quantum-secure-blockchain-preparing-for-the-era-of-quantum-computing/.

Computing power costs continue to decline, especially with advancements in processor architecture and cloud computing. People used to ask the question, "How much computing power do I need on my local system to deal with my peak processing needs?" Today, that thinking has shifted to "Can I quickly carve out in the cloud what I need temporarily for my most demanding computing workloads?" You pay for only what you use to accomplish a given task.

1.3 Java

The microprocessor revolution's most significant contribution to date is that it enabled the development of personal computers and smartphones. Microprocessors also have profoundly impacted intelligent consumer-electronic devices, including the enormous number of devices in the Internet of Things (IoT). Recognizing this early on, Sun Microsystems in 1991 funded an internal corporate research project led by James Gosling, which resulted in a C++-based object-oriented programming language that Sun called

Java.[21] Using Java, which is among the world's most senior and widely used high-level programming languages, you can write programs that will run on a great variety of computer systems and computer-controlled devices. This is sometimes called "write once, run anywhere (WORA)." This kind of portability is particularly powerful.

21. Paul Krill, "So Why Did They Decide to Call It Java?" InfoWorld, May 23, 2014. Accessed February 18, 2025. https://www.infoworld.com/article/2170044/so-why-did-they-decide-to-call-it-java-3.html.

Java drew the business community's attention because of the phenomenal interest in the Internet. It's now used to develop large-scale enterprise applications, enhance the functionality of web servers, provide applications for consumer devices (cell phones, smart-phones, cable TV boxes and more), develop robotics software and for many other purposes. It was the original language for developing Android smartphone and tablet apps—it's now one of two key Android development languages.[22] Sun Microsystems was acquired by Oracle in 2010.

22. "Kotlin (programming language)." Accessed February 18, 2025. https://en.wikipedia.org/wiki/Kotlin_(programming_language).

Java has been one of the most widely used general-purpose programming languages for three decades, so there is much existing Java code using older programming idioms—this is referred to as legacy code. We emphasize modern Java idioms in this book.

You may want to check out the curated list of top Java apps at

Click here to view code image

```
https://blogs.oracle.com/javamagazine/post/the-top-25-greatest-java-
   apps-ever-written
```

Also, we've integrated genAI exercises throughout the book beginning with this chapter's end-of-chapter Special Section: Generative AI Exercises. Exercise 1.2 asks you to prompt genAIs to list ways in which Java may be considered first among modern programming languages.

Java Release Cadence

Java follows a six-month release cadence that began with Java 9 in 2017. New versions are released every March and September,

enabling faster delivery of new language and library features. A release is designated as a Long-Term Support (LTS) version every two years. These are geared toward enterprises requiring stability and extended support. Non-LTS releases encourage experimentation with evolving innovative new features but have shorter support cycles.

JEPs, JSRs and the JCP

Throughout the book, we encourage you to research various aspects of Java online. Some acronyms you're likely to see are JEP, JSR and JCP.

Oracle uses **JEPs (JDK Enhancement Proposals)** to gather proposals from the Java community for changes to the Java language, APIs and tools, and to help create the road-maps for future Java Standard Edition (Java SE), Java Enterprise Edition (Java EE) and Java Micro Edition (Java ME) platform versions and the JSRs (Java Specification Requests) that define them. The complete list of JEPs can be found at

```
https://openjdk.org/jeps/0
```

JSRs (Java Specification Requests) are the formal descriptions of Java platform features' technical specifications. Each new feature has a JSR that goes through a review and approval process before it's added to Java. The complete list of JSRs can be found at

Click here to view code image

```
https://www.jcp.org/en/jsr/all
```

The **JCP (Java Community Process)** is responsible for developing JSRs. JCP expert groups create the JSRs and make them publicly available for review and feedback. You can learn more about the JCP at:

Click here to view code image

```
https://www.jcp.org/en/home/index
```

1.4 A Brief Review of Object Orientation

Building software quickly, correctly and economically remains an elusive goal at a time when demands for new and more powerful software are soaring. **Objects**, or more precisely—as we'll see in

Chapter 8—the **classes** objects come from, are essentially **reusable software components**. There are date objects, time objects, audio objects, video objects, automobile objects, people objects, etc. Almost any noun can be reasonably represented as a software object in terms of **attributes** (e.g., name, color and size) and **behaviors** (e.g., calculating, moving and communicating). Software developers have discovered that using a modular, object-oriented, design-and-implementation approach can make software development groups much more productive than was possible with earlier techniques—object-oriented programs are often easier to understand, test, debug, correct and evolve.

Automobile as an Object

Let's begin with a simple analogy. Suppose you want to drive a car and make it go faster by pressing its accelerator pedal. What must happen before you can do this? Well, someone has to **design** the car before you can drive it. A car typically begins as engineering drawings, similar to the **blueprints** that describe a house's design. These drawings include the design for an accelerator pedal. The pedal **hides** from the driver the complex mechanisms that make the car go faster, just as the brake pedal hides the mechanisms that slow the car, and the steering wheel hides the mechanisms that turn the car. This enables people with little or no knowledge of how engines, braking and steering mechanisms work to drive a car easily.

Before you can drive a car, it must be **built** from the engineering drawings that describe it. A completed car has an **actual** accelerator pedal to make the car go faster, but even that's not enough—the car won't accelerate on its own (hopefully!), so the driver must **press** the pedal to instruct the car to accelerate.

Methods and Classes

Let's use our car example to introduce some key object-oriented programming concepts. Performing a task in a program requires a **method**. The method houses the program statements that perform its task. It hides these statements from its user, just as a car's accelerator pedal hides from the driver the mechanisms that make the car go faster. In Java, we create a program unit called a **class** to hold the set of methods that perform the class's tasks. For example,

a class representing a bank account might contain a method to **deposit** money to an account, another to **withdraw** money from an account and a third to **query** the account's current balance. A class is similar to a car's engineering drawings, which house the design of an accelerator pedal, brake pedal, steering wheel, and so on.

Instantiation

Just as someone has to **build a car** from its engineering drawings before you can drive a car, you must **build an object** from a class before a program can perform the tasks defined by the class's methods. The process of doing this is called **instantiation**, and an object is then referred to as an **instance** of its class.

Reuse

Just as a car's engineering drawings can be **reused** many times to build many cars, you can **reuse** a class many times to build many objects of that class. Reusing existing classes when building new classes and programs saves time and effort. Reuse also helps you build more reliable and effective systems because existing classes and components often have been carefully **tested**, **debugged** and **performance-tuned**. Just as the notion of **interchangeable parts** was crucial to the Industrial Revolution, reusable classes are crucial to the software revolution spurred by object technology.

Messages and Method Calls

When you drive a car, pressing its gas pedal sends a **message** to the car to perform a task—that is, to "go faster." Similarly, you **send messages to an object**. Each message is implemented as a **method call** that tells a method of the object to perform its task. For example, a program might call a particular bank account object's **deposit** method to increase the account's balance by the deposit amount.

Attributes and Instance Variables

Besides having capabilities to accomplish tasks, a car also has **attributes**, such as its color, number of doors, amount of gas in its tank, current speed and record of total miles driven (i.e., its odometer reading). Like its capabilities, the car's attributes are represented as part of its design in its engineering diagrams (which, for example, include an odometer and a fuel gauge). As you drive a

car, these attributes are "carried along" with the car. Every car maintains its own attributes. For example, each car knows how much gas is in its own gas tank but not how much is in the tanks of other cars.

An object, similarly, has attributes that it carries along as it's used in a program. These attributes are specified as part of the object's class. For example, a bank account object has a **balance attribute** representing the amount of money in the account. Each bank account object knows the balance in the account it represents but not the balances of the other accounts in the bank. Attributes are specified by the class's **instance variables**.

Encapsulation and Information Hiding

Classes **encapsulate** (i.e., wrap) attributes and methods into objects created from those classes—an object's attributes and methods are intimately related. Objects may communicate with one another, but they're usually not allowed to know how other objects are implemented internally. Those details are hidden inside the objects. As we'll see, this **information hiding** is crucial to good software engineering.

Inheritance

A new class of objects can be created quickly and conveniently by **inheritance**. The new class absorbs the characteristics of an existing class, possibly customizing them and adding unique characteristics of its own. In our car analogy, an object of the class "convertible" certainly **is an** object of the more **general** class "automobile," but more **specifically**, the roof can be raised or lowered.

Interfaces

Java also supports **interfaces**—groups of method declarations that define *what* actions an object can perform without specifying *how* to perform them. In the car analogy, a "basic-driving-capabilities" interface consisting of a steering wheel, an accelerator pedal and a brake pedal would enable a driver to tell the car *what* to do. Once you know how to use this almost universal interface for turning, accelerating and braking, you can drive many types of cars, even though manufacturers may implement these systems differently.

A class **implements** zero or more interfaces, each of which can have methods, just as a car implements separate interfaces for basic

driving functions, controlling the radio, controlling the heating and air conditioning systems, and the like. Just as car manufacturers implement capabilities differently, classes may implement an interface's methods differently. For example, a software system may include a "backup" interface that offers the methods *save* and *restore*. Classes may implement those methods differently, depending on the types of things being backed up, such as programs, text, audios, videos, etc., and the types of devices where these items will be stored.

Object-Oriented Analysis and Design

Soon, you'll write Java programs. How will you create the code for your programs? Perhaps you'll turn on your computer and start entering code. This approach may work for small programs (like the ones we present in the book's early chapters), but what if you were asked to create a software system to control thousands of automated teller machines for a major bank? Or suppose you were asked to work on a team of thousands of software developers building the next generation of the U.S. air traffic control system? For large and complex projects such as these, you should not simply start entering programs.

To create the best solutions, you should follow a detailed analysis process for determining your project's **requirements** (i.e., defining *what* the system should do) and developing a **design** that satisfies them (i.e., deciding *how* the system should do it). Ideally, you'd go through this process and carefully review the design (and have it reviewed by others) before writing any code. If this process involves analyzing and designing your system from an object-oriented point of view, it's called an **object-oriented analysis and design (OOAD) process**. Java has **object-oriented programming (OOP)** capabilities that enable you to implement object-oriented designs as secure, high-performance, robust, working systems.

The UML (Unified Modeling Language)

Although many different OOAD processes exist, a single graphical language for communicating the results of any OOAD process has come into wide use. The **Unified Modeling Language (UML)** is the most widely used graphical scheme for modeling object-oriented systems. We present our first UML diagrams in Chapters 3 and 4, then use them in our deeper treatment of object-oriented programming through Chapter 10.

1.5 Java Application Programming Interface (API) and Open-Source Libraries

You can create each class, interface and method you need from scratch to form your programs. Instead, most Java programmers take advantage of the rich collections of existing classes, interfaces and methods in the **Java class libraries**, also known as the **Java API (Application Programming Interface)**. Thus, there are really two parts to learning Java programming:

- learning the Java language itself, and
- learning how to use the Java API's classes, interfaces and methods and those in other publicly available open-source libraries.

Using Java API classes and methods instead of writing your own versions can improve program reliability and performance because they're carefully written to work correctly and perform efficiently. This also shortens program development time.

Open Source Libraries

There are enormous numbers of open-source Java libraries that can help you perform significant tasks by writing only modest amounts of code. On **GitHub**, developers contribute to over 250,000 Java code repositories:

Click here to view code image

```
https://github.com/topics/java
```

In addition, pages such as

- https://github.com/akullpp/awesome-java
- https://github.com/Vedenin/useful-java-links
- https://github.com/mhrimaz/AwesomeJavaFX

provide curated lists of popular Java libraries and resources—you'll enjoy exploring these sites.

Building-Block Approach to Java Development

When programming in Java, you'll typically mix your own code with the following preexisting building blocks:

- Java API classes, interfaces and methods,
- open-source Java library classes, interfaces and methods,
- classes, interfaces and methods you create (after studying Chapters 8 and 9), and
- classes, interfaces and methods others have created and made available to you.

The advantage of creating your own classes, interfaces and methods is knowing exactly how they work. The disadvantage is the time-consuming effort to design, develop, debug and performance-tune them. Throughout the book, we use the existing Java APIs and various open-source libraries to leverage your programming efforts and **avoid "reinventing the wheel."** This is called **software reuse** and is a key part of our objects-natural teaching methodology.

1.6 Typical Java Program Development Environment

We now explain the steps to create and execute a Java program. Usually, there are five phases—**edit**, **compile**, **load**, **verify** and **execute**. We discuss them in the context of the Java Development Kit (JDK). See the Before You Begin that follows the Preface for information on downloading and installing the JDK on Windows and macOS.

Phase 1: Creating a Program

Phase 1 consists of editing a file with an editor program (or simply, an editor):
Using the editor, you type a Java program's source code, make any necessary corrections and save it to your system. Java source code files are given a name ending with the `.java` **extension**, indicating that the file contains Java source code.

Windows and macOS provide the basic editors Notepad and TextEdit, respectively. Some more powerful editors available online

include:

- jEdit (https://jedit.org; cross-platform, written in Java),
- Sublime (https://www.sublimetext.com; cross-platform)
- Notepad++ (https://notepad-plus-plus.org; Windows-only) and
- TextMate (https://textpad.com; macOS-only)

Integrated development environments (IDEs) provide tools that support the software development process, such as editors, debuggers for locating **logic errors** that cause programs to execute incorrectly, and more. The most popular Java IDEs are:

- JetBrains' IntelliJ IDEA Community Edition,
- the Eclipse Foundation's Eclipse IDE for Java Developers and
- Microsoft's Visual Studio Code cross-platform code editor—this is technically a powerful code editor but has plugins for Java development (and for many other programming languages).

We discuss installing each in the Before You Begin section. On the book's website at

```
https://deitel.com/javafp5
```

we provide test-drives showing how to execute this book's Java programs and how to develop new Java programs with these IDEs.

Phase 2: Compiling a Java Program into Bytecodes

In Phase 2, you use the **`javac` command** (the **Java compiler**) to **compile** a program:
For example, to compile a program called `Welcome.java`, you'd type

```
javac Welcome.java
```

in your system's command window (Section 1.7). If the program compiles, the compiler produces a **`.class` file** called `Welcome.class`. IDEs typically do this for you. If the compiler detects errors, you must go back to Phase 1 and correct them. In Chapter 2, we'll discuss the kinds of errors the compiler can detect.

When using `javac`, if you receive a message such as "`bad command or filename`," "`javac: command not found`" or "`'javac' is`

`not recognized as an internal or external command, operable program or batch file`," your JDK is not installed correctly. Carefully review the installation instructions in the Before You Begin.

The Java compiler translates Java source code into **bytecodes** representing the tasks to execute in the execution phase (Phase 5). The **Java Virtual Machine** (**JVM**)—part of the JDK and the Java platform's foundation—executes bytecodes.[23] A **virtual machine** (**VM**) is software that simulates a computer but hides the underlying operating system and hardware from the programs that interact with it. If the same VM is implemented on many computer platforms, applications written for that type of VM can be used portably on all those platforms. The JVM is one of the most widely used virtual machines.

23. "Java virtual machine." Wikipedia. Wikimedia Foundation. Accessed February 18, 2025. https://en.wikipedia.org/wiki/Java_virtual_machine#Bytecode_interpreter_and_just-in-time_compiler.

Unlike **platform-dependent** machine-language instructions, which are dependent on specific computer hardware, bytecode instructions are **platform-independent**. So, Java's bytecodes are **portable**—without recompiling the source code, the same bytecode instructions can execute on any platform containing a JVM that understands the version of Java in which the bytecodes were compiled. The JVM is invoked by the **`java` command**. For example, to execute a Java program called `Welcome`, you'd type the command

```
java Welcome
```

in a command window to invoke the JVM, which would then initiate the steps necessary to execute the application. This begins Phase 3. IDEs typically provide a menu item, such as Run, that invokes the `java` command for you.

Phase 3: Loading a Program into Memory

In Phase 3, the JVM places the program in memory to execute it—this is known as **loading:**
The JVM's **class loader** takes the `.class` files containing the program's bytecodes and transfers them to primary memory. It also loads any of the `.class` files provided by Java that your program uses.

Phase 4: Bytecode Verification

In Phase 4, as the classes are loaded, the **bytecode verifier** examines their bytecodes to ensure that they're valid and do not violate Java's security restrictions:

Phase 5: Execution

In Phase 5, the JVM **executes** the bytecodes to perform the program's specified actions:
In early Java versions, the JVM interpreted bytecodes. Most programs would execute slowly because the JVM would interpret and execute one bytecode at a time. Some modern computer architectures can execute several instructions in parallel. Today's JVMs typically execute bytecodes using a combination of interpretation and **just-in-time (JIT) compilation**. In this process, the JVM analyzes the bytecodes as they're interpreted, searching for hot spots—bytecodes that execute frequently. For these parts, a **just-in-time** (**JIT**) **compiler**, such as the OpenJDK's **HotSpot compiler**, translates the bytecodes into the computer's machine language. When the JVM encounters these compiled parts again, the faster machine-language code executes. Thus, programs actually go through two compilation phases—one in which Java code is translated into bytecodes (for portability across JVMs on different computer platforms) and a second in which, during execution, the bytecodes are translated into machine language for the computer on which the program executes.

Problems That May Occur at Execution Time

Programs might not work on the first try. Each preceding phase could fail due to various errors discussed throughout this book. For example, an executing program might try to divide by zero—an illegal operation for whole-number arithmetic in Java. This would cause the Java program to display an error message. If this occurred, you'd return to the edit phase, make the necessary corrections and proceed through the remaining phases again to determine whether the corrections fixed the problem(s).

Errors such as division by zero occur as a program runs, so they're called **runtime errors** or **execution-time errors**. **Fatal runtime errors** cause programs to terminate immediately without successfully performing their jobs. **Nonfatal runtime errors** allow programs to run to completion, often producing incorrect results.

1.7 Test-Driving a Java Program with the Java Development Kit (JDK)

Before reading this test-drive, read the Before You Begin that follows the Preface to ensure you've placed the book's examples in your user account's `Documents/examples` folder and installed the JDK. In this section, you'll use the JDK's tools to compile and run your first Java program[24]—a guess-the-number game, which picks a random number from 1 to 1,000 and prompts you to guess it. If you guess the randomly selected number, the game ends. If you guess incorrectly, the program indicates whether your guess is higher or lower than the correct number. You keep guessing until you guess the number. There's no limit on the number of guesses you can make, but it should not take more than 10 if you use the binary search technique.

24. We intentionally do not cover this Java program's code here. This section's purpose is to demonstrate compiling and running a program, so you can execute our examples in subsequent chapters.

1.7.1 Compiling and Running the Program with `javac` and `java`

We'll show two ways to run the program. Here, we focus on the traditional approach in which you

- use the `javac` command to compile the program, then
- use the `java` command to run it.

IDEs typically provide a convenient way to perform both steps by pressing (or selecting) a button, selecting a menu item or using a keyboard shortcut.

Step 1: Changing to the `ch01` Folder

First, open a terminal (command-line) window on your system:

- On macOS, select Utilities in the Finder's Go menu, then open the Terminal application.
- On Windows, enter `Terminal` in your Windows taskbar's Search field, then select Terminal in the search results to open a Windows PowerShell window.

Next, use the `cd` command to change to the `ch01` subfolder of the book's `examples` folder:

- In the macOS Terminal window, enter the following command and press (or select) *Enter*:

```
cd ~/Documents/examples/ch01
```

- In the Windows PowerShell window, enter the following command and press (or select) *Enter*:

```
cd ~\Documents\examples\ch01
```

In the remaining steps, we'll show the macOS Terminal application's prompts—the commands for compiling and running the program using the JDK's command-line tools and your interactions with the program are identical between macOS and Windows.

Step 2: Compiling the Program

Before running the guess-the-number game, you must compile its Java source code with the `javac` command:

Click here to view code image

```
pauldeitel@MacBook-Pro ch01 % javac GuessTheNumberGame.java
pauldeitel@MacBook-Pro ch01 %
```

If the program compiles without errors, the `javac` command creates the `.class` file

```
GuessTheNumberGame.class
```

containing the program's bytecodes.

Step 3: Running the Program

To run the program, use the `java` command followed by the name of the `.class` file without the `.class` extension:

Click here to view code image

```
(base) pauldeitel@MacBook-Pro ch01 % java GuessTheNumberGame
I have a number between 1 and 1000.
Can you guess my number?
Please enter your first guess.
?
```

The application displays `"Please enter your first guess."`, then displays a question mark (`?`) as a prompt on the next line.

Step 4: Entering Your First Guess

At the prompt, enter **`500`** and press (or select) *Enter*:

Click here to view code image

```
(base) pauldeitel@MacBook-Pro ch01 % java GuessTheNumberGame
I have a number between 1 and 1000.
Can you guess my number?
Please enter your first guess.
? 500
Too low. Try again.
?
```

The program displayed `"Too low. Try again."` because the value entered was less than the program's chosen number.[25]

25. We wrote this program so it chooses the same "random" value each time it is run, so you'll be able to use the same guesses we used.

Step 5: Entering Another Guess

At the next prompt, type **`750`** and press (or select) *Enter*:

Click here to view code image

```
(base) pauldeitel@MacBook-Pro ch01 % java GuessTheNumberGame
I have a number between 1 and 1000.
Can you guess my number?
Please enter your first guess.
? 500
Too low. Try again.
? 750
Too high. Try again.
?
```

The application displayed `"Too high. Try again."` because the value was greater than the program's chosen number.

Step 6: Entering Additional Guesses

Continue to play the game by entering values until you guess the correct number. When you guess correctly, the application displays `"Excellent! You guessed the number."`:

Click here to view code image

```
? 625
Too low. Try again.
? 687
Too high. Try again.
? 656
Too high. Try again.
? 640
Too low. Try again.
? 648
Too high. Try again.
? 644
Too high. Try again.
? 642

Excellent! You guessed the number!
Would you like to play again (y or n)? n
```

Step 7: Playing the Game Again or Exiting the Program

After guessing the correct number, the application asks if you want to play again. Entering `y` causes the application to choose a new number and start a new game. Entering `n` terminates the application and returns you to the command-line prompt.

10 or Fewer Guesses

Guessing a number from 1 to 1000 should take at most 10 tries. When you start with 500 (assuming it is not the correct answer), you immediately eliminate half the numbers as your guess is either higher or lower than the correct answer. In the test-drive, 500 was too low, so we eliminated 1–500, leaving 500 values (501–1000) after one guess. Each guess should be the halfway point in the remaining range. On the next guess, 750 was too high, so we eliminated 750–1000, narrowing the potential range to 501–749, which is about half of the 500 remaining from the previous guess. If you repeatedly choose the halfway point in the remaining range, ten guesses would reduce the range to one value, which would be the correct answer. This halving technique is a property of the binary search algorithm.

1.7.2 Running the Program's Source-Code

File Directly with the `java` Command

You can run a single-source-file program—like Section 1.7.1's guess-the-number game—directly using the `java` command, as in:[26]

```
java GuessTheNumberGame.java
```

26. "JEP 330: Launch Single-File Source-Code Programs." Accessed February 18, 2025. https://openjdk.org/jeps/330.

When you execute the source code directly, the `java` command compiles the program in memory and immediately runs it. Unlike `javac`, the `java` command will not store a `.class` file containing the program's bytecodes on your system.

You can even run multiple-source-file programs,[27] like those you'll begin building in **Chapter 8**, **Real-World Modeling with Custom Classes**. In this case, all the program's source files are compiled in memory, then the program executes. These features make running a Java program more convenient for quick testing purposes.

27. "JEP 458: Launch Multi-File Source-Code Programs." Accessed February 18, 2025. https://openjdk.org/jeps/458.

1.8 Internet, World Wide Web, the Cloud and IoT

In the late 1960s, ARPA—the Advanced Research Projects Agency of the United States Department of Defense—rolled out plans for networking the main computer systems of approximately a dozen ARPA-funded universities and research institutions. The computers were to be connected with communications lines operating at speeds on the order of 50,000 bits per second, a stunning rate at a time when most people (of the few who even had networking access) were connecting over telephone lines to computers at a rate of 110 bits per second. Academic research was about to take a giant leap forward. ARPA implemented what quickly became known as the ARPANET, the precursor to the **Internet**. Today's fastest consumer Internet speeds are on the order of billions of bits (gigabits) per second. The current Internet speed record holder is Japan's National Institute of Information and Communications Technology, which achieved download speeds of 402 terabits (trillions of bits) per second—about 1.6 million times more than the average U.S. consumer's 250 megabits per second broadband

speed.[28]

28. Paulius Grinkevičius, “Japan Breaks Internet Speed Record, Hitting 402 Terabits per Second," CyberNews, July 4, 2024. Accessed February 18, 2025. https://cybernews.com/tech/internet-connection-speed-world-record.

Things worked out differently from the original plan. Although the ARPANET enabled researchers to network their computers, its main benefit proved to be the capability for quick and easy communication via what came to be known as electronic mail (e-mail). This is true even on today’s Internet, with e-mail, instant messaging, file transfer and social media, such as YouTube, Facebook, WhatsApp, Instagram, Snapchat, TikTok, Reddit and X (formerly Twitter), enabling billions of people worldwide to communicate quickly and easily.

The protocol (set of rules) allowing computers to communicate with one another over the ARPANET became known as the **Transmission Control Protocol (TCP)**. It ensured messages, consisting of sequentially numbered pieces called **packets**, were properly delivered from sender to receiver, arrived intact and reassembled in the correct order.

1.8.1 The Internet: A Network of Networks

In parallel with ARPANET’s evolution, organizations worldwide implemented their own networks for intra- and inter-organization communication. A huge variety of networking hardware and software appeared. One challenge was to enable these different networks to communicate with each other. ARPA accomplished this by developing the **Internet Protocol (IP)**, which created a “network of networks,” the Internet’s current architecture. The combined set of protocols is now called **TCP/IP**. Each Internet-connected device has an **IP address**—a unique numerical identifier used by devices communicating via TCP/IP to locate one another on the Internet.

Companies rapidly realized that they could use the Internet to improve their operations and offer new and better client services. So, they spent large amounts of money developing and enhancing their Internet presence. This generated intense competition among communications carriers and hardware and software suppliers to meet the increased infrastructure demand. As a result, **Internet bandwidth**—the information-carrying capacity of communications lines—increased tremendously while hardware costs plummeted.

1.8.2 The World Wide Web: Making the Internet User-Friendly

The **World Wide Web** (or simply "the web") is a collection of hardware and software associated with the Internet that allows computer users to locate and view documents (with various combinations of text, graphics, animations, audios and videos) on almost any subject. In 1989, Tim Berners-Lee of CERN (the European Organization for Nuclear Research) began developing the **HyperText Markup Language (HTML)**—the technology for sharing information via "hyperlinked" text documents. He also wrote communication protocols such as the **HyperText Transfer Protocol (HTTP)** to form the backbone of his new hypertext information system, which he called the World Wide Web.

In 1994, Berners-Lee founded the **World Wide Web Consortium (W3C; `w3.org`)**, which is devoted to developing web technologies. A key W3C goal is to make the web universally accessible to everyone regardless of ability, language or culture.

1.8.3 The Cloud

More and more computing today happens "in the cloud" using software and data distributed across the Internet worldwide rather than locally on your computer or mobile device. **Cloud computing** allows you to increase or decrease computing resources to meet your needs at any given time, which is more cost-effective than purchasing your own hardware to provide enough storage and processing power to meet occasional peak demands. Cloud computing also saves money by shifting to the service provider the burden of managing computing resources (such as installing and upgrading the software, security, backups and disaster recovery capabilities).

The apps you use daily are heavily dependent on **cloud-based services**. These use massive clusters of computing resources (computers, processors, memory, storage, etc.) and databases that communicate over the Internet with each other and the apps you use. A service that provides access to itself over the Internet is known as a **web service**. As of November 2024, Amazon Web Services (AWS; 32% market share), Microsoft Azure (20%) and Google Cloud Platform (GCP; 12%) account for 64% of the cloud services market.[29]

29. Statista, "Worldwide Market Share of Leading Cloud Infrastructure

Service Providers." November 1, 2024. Accessed February 18, 2025, https://www.statista.com/chart/18819/worldwide-market-share-of-leading-cloud-infrastructure-service-providers/.

1.8.4 The Internet of Things (IoT)

The Internet is not just a network of computers—it's an **Internet of Things (IoT)**. A "thing" is any device with an IP address and the ability to send and sometimes receive data automatically over the Internet. Such "things" include:

- cars with transponders for paying tolls and GPS for navigation,
- autonomous robots,
- monitors for parking space availability in a garage,
- heart monitors implanted in humans,
- smart sensors that monitor energy usage, weather, water quality and traffic,
- smoke and carbon monoxide detectors,
- radiation detectors,
- item trackers in warehouses,
- mobile apps that can track your movements,
- thermostats that adjust room temperatures based on weather forecasts and activity in the home,
- wearables like smartwatches, fitness trackers and smart glasses,
- intelligent home appliances, lighting, doorbells and locks, and
- many more.

There are expected to be 41.6 billion active IoT devices in 2025.[30]

30. Dell Technologies, "Internet of Things and Data Placement," Edge to Core and the Internet of Things, Accessed February 18, 2025, https://infohub.delltechnologies.com/en-us/l/edge-to-core-and-the-internet-of-things-2/internet-of-things-and-data-placement/.

1.8.5 Edge Computing

Many IoT devices are used in **edge computing**,[31] in which data processing, analysis and storage occur on or close to the devices that produce the data—that is, at the network's edge. It "allows for faster processing and real-time decision-making, which is critical for applications requiring immediate responses, such as IoT devices and autonomous vehicles or edge computers."[32] For example, if a self-driving car loses its Internet connection, it must still make the decisions that enable it to drive safely. Edge computing is used in many fields, including healthcare, manufacturing, retail, clean energy, smart power grids, transportation, autonomous vehicles, smart cities and infrastructure, industrial automation and process control, defense, aerospace, agriculture, gaming, virtual reality, video surveillance and more.[33,34]

31. "Edge computing." Wikipedia. Wikimedia Foundation. Accessed February 14, 2025. https://en.wikipedia.org/wiki/Edge_computing.

32. Advantech, "Edge Computing: Enhancing Real-Time Data Processing and Decision-Making," September 26, 2024. Accessed February 14, 2025. https://www.advantech.com/en-us/resources/industry-focus/edge-computing.

33. Corvalent. "7 Essential Edge Computing Use Cases for Today's Industries." Accessed February 18, 2025. https://corvalent.com/news/7-essential-edge-computing-use-cases-for-industries/.

34. Daffodil Software. "Top 24 Industrial Applications of Edge Computing," April 3, 2023. Accessed February 18, 2025. https://insights.daffodilsw.com/blog/applications-of-edge-computing.

1.8.6 Mashups

The applications-development methodology of **mashups** enables you to rapidly develop powerful software by combining (often free) complementary web services and various forms of information feeds. One of the first mashups, `housingmaps.com`, combined the real-estate listings from `craigslist.org` with Google Maps to show the locations of homes for sale or rent in a given area.

A focus of this book's object-natural approach is major software reuse beginning in Chapter 2—you'll work with easy-to-use, powerful preexisting classes that do significant things. Similarly, before building your own web services, consider reusing the many powerful (and often free) preexisting services available across the Internet. Numerous online catalogs list dozens to thousands of existing web services:

- **GitHub's Public APIs page** (https://github.com/public-apis/public-apis) lists 51 categories containing hundreds of publicly available web services.
- **The Google APIs Explorer** (https://developers.google.com/apis-explorer) lets you learn about and experiment with almost 300 Google web services without writing code.
- **The Free Public APIs page** (https://www.freepublicapis.com/) lists hundreds of free public APIs.
- **Rapid** (https://rapidapi.com35) claims to be the world's largest API hub, listing thousands of free and paid APIs. Rapid also provides tools to help companies build and distribute their own APIs.

35. Nokia. "Nokia Acquires Rapid Technology and R&D Unit to Strengthen Development of Network API Solutions and Ecosystem." November 13, 2024. Accessed February 18, 2025. https://www.nokia.com/about-us/news/releases/2024/11/13/nokia-acquires-rapid-technology-and-rd-unit-to-strengthen-development-of-network-api-solutions-and-ecosystem/.

1.9 Metaverse

The term Metaverse was coined in Neal Stephenson's 1992 novel *Snow Crash*, where the character Hiro "exists in a computer-generated universe that his computer is drawing into his goggles and pumping into his earphones. …this imaginary place is known as the Metaverse."[36,37] Gartner says the Metaverse is the Internet's next iteration and defines it as a "collective virtual shared space, created by the convergence of virtually enhanced physical and digital reality."[38] This section presents several key Metaverse technologies. Section 1.13 presents artificial intelligence, which improves the Metaverse's immersive nature by helping generate realistic digital avatars, environments, natural-language dialog and more.[39]

36. Neal Stephenson. *Snow Crash*. Bantam Books, 1993. p. 24.
37. "Metaverse." Wikipedia. Wikimedia Foundation. Accessed February 13, 2025. https://en.wikipedia.org/wiki/Metaverse.
38. Jackie Wiles, "What Is a Metaverse? And Should You Be Buying In?" October 21, 2022. Accessed February 18, 2025. https://www.gartner.com/en/articles/what-is-a-metaverse.
39. "The Role of Artificial Intelligence in the Metaverse," IEEE Metaverse.

Accessed February 18, 2025. https://metaversereality.ieee.org/publications/articles/the-role-of-artificial-intelligence-in-the-metaverse.

1.9.1 Virtual Reality (VR)

Virtual reality (VR) typically refers to simulated, three-dimensional (3D) environments that you interact with using:

- **non-immersive VR**—in which the 3D environment is viewed in two-dimensional space using a computer's screen,
- **semi-immersive VR**—in which the 3D environment is viewed in two-dimensional space using large displays, such as big-screen TVs or video walls, or
- **fully immersive VR**—in which users interact with the 3D environment via a head-mounted display (like a headset) and other devices (like hand-held controllers), making them feel like they're part of the environment.

Some popular virtual reality devices include Meta Quest 3, Meta Quest Pro, Sony Play-Station VR2 and HTC Vive XR Elite.[40]

40. Will Greenwald, "The Best VR Headsets," PCMag. February 5, 2025. Accessed February 18, 2025, https://www.pcmag.com/picks/the-best-vr-headsets.

1.9.2 Augmented Reality (AR)

Augmented reality (AR) projects real-time computer-generated content—such as images, sound, video, animation and more—onto what is seen in the real world, typically via devices with built-in cameras, like smartphones, tablets and computerized glasses that enable both the real world and virtual content to be seen at the same time.[41,42]

41. Alexander S. Gillis, "What is augmented reality (AR)?" March 2024. Accessed February 18, 2025. https://www.techtarget.com/whatis/definition/augmented-reality-AR.

42. "Augmented reality." Wikipedia. Wikimedia Foundation. Accessed February 18, 2025. https://en.wikipedia.org/wiki/Augmented_reality.

One of the most popular examples of augmented reality is the popular mobile game app Pokémon Go, in which Pokémon characters are projected onto real-world locations that you find via

the GPS in your mobile devices. A decade after its release, Niantic continues to find new ways to keep Pokemon Go players engaged.[43]

43. "MLB 2025," Pokémon GO Live. February 12, 2025. Accessed February 18, 2025. https://pokemongolive.com/post/mlb-2025.

Other uses include:

- In sports like baseball, golf and hockey, broadcasts help you follow the baseball, golf ball or puck by displaying a line that traces the path of the ball or puck.
- National Football League broadcasts display colored lines over the field, showing the line of scrimmage and where the team needs to reach for a first down.
- Navigation apps can display real-time navigation and traffic information on real-world maps and street-level views.
- Home decoration apps can show you what a room will look like after you repaint it or add specific furniture items.
- Clothing retailers often can show you what you'd look like in a particular clothing item you're considering purchasing.
- Museums can make exhibits more interactive by displaying information about a specific exhibit you are viewing.

Many companies have or are working on **AR glasses and headsets**.[44,45] Google, Samsung and Qualcomm are collaborating on **AndroidXR**—an Android mobile operating system extension that combines the digital and physical worlds—and corresponding computer chips and AI-powered glasses and headsets.[46]

44. Prakhar Khanna and Kerry Wan, "CES 2025: The 8 Most Advanced Smart Glasses We Tried and Were Impressed By." ZDNet. January 10, 2025. Accessed February 18, 2025. https://www.zdnet.com/article/ces-2025-the-8-most-advanced-smart-glasses-we-tried-and-were-impressed-by/.

45. Will Greenwald, "The Best Smart Glasses," PCMag, January 16, 2025. Accessed February 18, 2025. https://www.pcmag.com/picks/the-best-smart-glasses.

46. Scott Stein, "I Tried Google and Samsung's Next-Gen Android XR Headsets and Glasses, and the Killer App Is AI," CNET. December 12, 2024. Accessed February 18, 2025. https://www.cnet.com/tech/computing/i-tried-google-and-samsungs-next-gen-androidxr-headsets-and-glasses-and-the-killer-app-is-ai/.

1.9.3 Mixed Reality

Mixed reality is similar to augmented reality but takes it further by enabling you to interact with the virtual objects projected into your real-world view. In that sense, mixed reality combines aspects of augmented and virtual reality. Like virtual reality, the best mixed-reality experiences typically require a headset.[47,48] **Extended reality (XR)** encompasses virtual, augmented and mixed reality.[49]

47. Bernard Marr, "The Important Difference Between Augmented Reality And Mixed Reality." Accessed February 18, 2025. https://bernardmarr.com/the-important-difference-between-augmented-reality-and-mixed-reality/.

48. "Mixed reality." Wikipedia. Wikimedia Foundation. Accessed February 18, 2025. https://en.wikipedia.org/wiki/Mixed_reality.

49. "Extended reality." Wikipedia. Wikimedia Foundation. Accessed February 18, 2025. https://en.wikipedia.org/wiki/Extended_reality.

1.9.4 Blockchain

Accounting systems use ledgers to keep track of transactions. **Blockchain**[50,51] is a software-based ledger distributed over nodes on the Internet. It maintains transaction data in cryptographically secured chunks—called **blocks**. These blocks are linked together, and new blocks are always added at the end. Once created, each block is **immutable** (not modifiable), so the blockchain maintains a permanent transaction record. The blockchain is decentralized, so no one person or group has control. This decentralization is the basis for the **decentralized apps (dApps)** that enable digital ownership in the Metaverse.

50. "Blockchain." Wikipedia. Wikimedia Foundation. Accessed February 18, 2025. https://en.wikipedia.org/wiki/Blockchain.

51. Adam Hayes, "Blockchain Facts: What Is It, How It Works, and How It Can Be Used," September 16, 2024. Accessed February 18, 2025. https://www.investopedia.com/terms/b/blockchain.asp.

Blockchain is behind various digital technologies that have skyrocketed in popularity over the last several years, such as cryptocurrencies (Section 1.9.5) and non-fungible tokens (Section 1.9.7). It's also used to implement many other kinds of applications,[52] including

- decentralized finance (DeFi) applications (lending, money transfer),
- smart contracts,

- safer Internet of Things (IoT) apps,
- secure digital identities to help prevent fraud,
- securing and sharing healthcare information,
- business logistics, and
- protecting intellectual property (IP).

52. Sam Daley, "33 Blockchain Applications and Real-World Use Cases." March 2, 2023. Accessed February 18, 2025. https://builtin.com/blockchain/blockchain-applications.

1.9.5 Bitcoin and Cryptocurrency

"A **cryptocurrency** is a digital or virtual currency secured by cryptography."[53] Cryptocurrency (or crypto) is stored in digital wallets and generally uses blockchain technology for secure, decentralized transaction tracking and processing.[54] The original, most well-known and most valuable cryptocurrency is Bitcoin, which was created in 2009.[55] It is a decentralized cryptocurrency built using blockchain technology.[56]

53. Jake Frankenfield, "Cryptocurrency Explained With Pros and Cons for Investment." February 4, 2023. Accessed February 18, 2025. https://www.investopedia.com/terms/c/cryptocurrency.asp.
54. "Cryptocurrency." Wikipedia. Wikimedia Foundation. Accessed February 18, 2025. https://en.wikipedia.org/wiki/Cryptocurrency.
55. Julie Pinkerton, "The History of Bitcoin, the First Cryptocurrency." February 27, 2023. Accessed February 18, 2025. https://money.usnews.com/investing/articles/the-history-of-bitcoin.
56. "Bitcoin." Wikipedia. Wikimedia Foundation. Accessed February 18, 2025. https://en.wikipedia.org/wiki/Bitcoin.

1.9.6 Ethereum

Ethereum is a programmable platform that enables you to "build and deploy decentralized applications (dApps) on its network … that use the blockchain to store data or control what your app can do." One of its key features is **smart contracts**, which are used to implement lending apps, decentralized trading exchanges, insurance, crowdfunding apps and more.[57] Its native cryptocurrency is **Ether**—the world's second most valuable cryptocurrency.[58]

57. “What is Ethereum?” Last updated February 12, 2025. Accessed February 18, 2025. https://ethereum.org/en/what-is-ethereum/.

58. “Ethereum.” Wikipedia. Wikimedia Foundation. Accessed February 18, 2025. https://en.wikipedia.org/wiki/Ethereum.

1.9.7 Non-Fungible Tokens (NFTs)

Non-fungible tokens (NFTs)[59] are blockchain-based “cryptographically unique tokens that are linked to digital or physical content, providing proof of ownership.” They’re commonly used to enable individuals to own unique items, such as digital or physical artwork and photographs; limited edition digital collectibles; virtual real estate and goods in games and the Metaverse; physical real estate; intellectual property; and digital books.[60]

59. “Non-fungible token.” Wikipedia. Wikimedia Foundation. Accessed February 18, 2025. https://en.wikipedia.org/wiki/Non-fungible_token.

60. Melanie Kramer, Stephen Graves and Daniel Phillips, “Beginner’s Guide to NFTs: What Are Non-Fungible Tokens?” Jan 12, 2022. Accessed February 18, 2025. https://decrypt.co/resources/non-fungible-tokens-nfts-explained-guide-learn-blockchain.

Some artwork NFTs have sold for massive amounts of money:[61]

- The most expensive NFT ever sold was Pak’s “The Merge,” purchased by 29,983 individual buyers for $91.8 million—each buyer owns a small portion of the NFT.
- The most expensive NFT ever sold to one buyer was “Everydays: The First 5000 Days” by digital artist Mike “Beeple” Winkelmann, which sold for $69.3 million.

61. Casey Anderson, “Most Expensive NFT,” NFT Evening. December 30, 2024. Accessed February 18, 2025. https://nftevening.com/most-expensive-nft/.

1.9.8 Web3

Like the Metaverse, some people refer to **Web3** as the Internet’s next generation.[62][63] Web3 and the Metaverse are sometimes confused with one another because they share some of the same technologies. Forbes defines Web3 as “the decentralized Internet—built on distributed technologies like blockchain … rather than

centralized on servers owned by individuals or corporations."[64] A key focus of Web3 is that blockchain and its related technologies (like NFTs, cryptocurrencies and smart contracts) will give users greater control of their data and online interactions.

62. David Essex, Sean Michael Kerner, and Alexander S. Gillis, "What Is Web 3.0 (Web3)? Definition, Guide and History," TechTarget. Accessed February 18, 2025. https://www.techtarget.com/whatis/definition/Web-30.

63. Lauren Gibbons Paul, "Top Web 3.0 Trends and Predictions for 2025 and Beyond," TechTarget, January 3, 2025. Accessed February 18, 2025. https://www.techtarget.com/searchcio/tip/Top-Web-30-trends-and-predictions.

64. Bernard Marr, "The Important Difference Between Web3 And The Metaverse." February 22, 2022. Accessed February 18, 2025. https://www.forbes.com/sites/bernardmarr/2022/02/22/the-important-difference-between-web3-and-the-metaverse/.

1.10 Software Development Technologies

As you develop software, you'll frequently encounter the following buzzwords:

- **Refactoring**: Reworking programs to make them clearer and easier to maintain while preserving their correctness and functionality and sometimes improving performance. Many IDEs contain built-in refactoring tools to automatically do significant portions of the reworking.[65]

  65. "Code refactoring." Wikipedia. Wikimedia Foundation. Accessed February 18, 2025. https://en.wikipedia.org/wiki/Code_refactoring.

- **Design patterns**: Proven patterns for constructing flexible and maintainable object-oriented software. The field of design patterns tries to enumerate those recurring patterns, encouraging software designers to reuse them to develop better-quality software faster, with less money, time and effort.[66]

  66. "Software design pattern." Wikipedia. Wikimedia Foundation. Accessed February 18, 2025. https://en.wikipedia.org/wiki/Software_design_pattern.

- **Software Development Kits (SDKs)**: The tools, libraries and documentation developers use to program applications. SDKs can help you create applications for specific operating

systems, gaming platforms, cloud platforms and more.[67] The Java Development Kit (JDK) is an example of an SDK.

67. "Software development kit." Wikipedia. Wikimedia Foundation. Accessed February 18, 2025. https://en.wikipedia.org/wiki/Software_development_kit.

- **Version Control Systems (VCS)**: Tools that enable you to manage and keep track of the changes you make in software as you develop it over time. **Git**—the technology behind GitHub—is by far the most widely used version control system, with 93% of developers using it.[68]

68. GitHub, "The State of Open Source and AI," GitHub Blog, Updated July 30, 2024. Accessed February 18, 2025. https://github.blog/news-insights/research/the-state-of-open-source-and-ai/.

- **Agile software development**: A way to create software in small steps, where development teams regularly improve the software based on feedback from customers who use parts of the software before it's finished. This approach helps teams quickly adapt when needs or goals change.[69]

69. "Agile software development." Wikipedia. Wikimedia Foundation. Accessed February 18, 2025. https://en.wikipedia.org/wiki/Agile_software_development.

- **Test-Driven Development (TDD)**: An aspect of agile software development in which developers create software test-cases before creating the software itself. Then, as they develop the software, they continually ensure it passes all the test cases, enabling them to confirm the software meets the system requirements.[70]

70. "Test-driven development." Wikipedia. Wikimedia Foundation. Accessed February 18, 2025. https://en.wikipedia.org/wiki/Test-driven_development.

- **Microservices**: A software architecture where an application is built from many small, independent services that work together. Each microservice focuses on a specific function. Microservices are commonly implemented as web services.[71] For example, the Netflix streaming service uses microservices for video playback, recommendations, content discovery, user authentication, billing, and more.[72] Though Netflix does not allow public use of its microservices, it does open-source aspects of its software and development architecture on the Netflix Open Source Software Center at

https://netflix.github.io/

71. “Microservices.” Wikipedia. Wikimedia Foundation. Accessed February 18, 2025. https://en.wikipedia.org/wiki/Microservices.

72. “How Many Microservices Are There in Netflix?” GeeksforGeeks, March 7, 2024. Accessed February 18, 2025. https://www.geeksforgeeks.org/how-many-microservices-are-there-in-netflix/.

- **Containerization**: A technique of packaging software into containers—lightweight, portable units that contain everything needed to run an application, such as code, libraries and dependencies. Containerization hides the underlying platform details using a technique called virtualization, enabling containers to execute consistently across platforms.[73] Docker is the most popular containerization platform.[74]

73. “Containerization (computing).” Wikipedia. Wikimedia Foundation. Accessed February 18, 2025. https://en.wikipedia.org/wiki/Containerization_(computing).

74. “Containerization.” Accessed February 18, 2025. https://6sense.com/tech/containerization.

1.11 Data Analytics and Data Science

We formed this section by asking genAIs to define data analytics and data science, then edited and used Gemini’s response as the basis for this section.[75]

75. *Gemini* response to “Consider the terms statistics, data analytics and data science. Precisely define the terms ‘data analytics’ and ‘data science’ and carefully explain the distinction between the two.” December 1, 2024. https://gemini.google.com. We edited the response.

Statistics

Statistics involves the study of collecting, organizing, analyzing, interpreting and presenting data. It provides the mathematical and theoretical framework for data analytics and data science.

Data Analytics

Data analytics is the process of examining existing datasets to uncover trends, patterns and insights. It focuses on understanding past data to inform present decisions, using techniques like descriptive statistics, data visualization and business intelligence. Essentially, data analytics aims to answer questions like "What happened?", "Why did it happen?" and "What can be done to improve?" For example, analyzing website traffic data to understand user behavior and optimize website design falls under data analytics.

Data Science

Data science is a multidisciplinary field that uses scientific methods, algorithms, processes, and systems to extract knowledge and insights[76] from both structured and unstructured data. It encompasses techniques such as machine learning, statistical modeling, data mining and predictive analytics. Data science aims to answer questions like "What might happen in the future?", "How can we optimize outcomes?" and "What new discoveries can we make from this data?" It often involves building models to predict future outcomes and automate decision-making. For instance, data science might be used to develop a machine-learning model that predicts customer churn based on historical data.

76. Datamites Team, "How to Become a Data Scientist in Chandigarh," DataMites (blog), September 8, 2023. Accessed February 18, 2025. https://datamites.com/blog/how-to-become-a-data-scientist-in-chandigarh/.

Distinction Between Data Analytics and Data Science

The primary distinction between data analytics and data science lies in their focus and methods:

- Data analytics focuses on understanding past data using descriptive statistics and visualization.
- Data science focuses on predicting future outcomes and discovering new insights by employing a broader range of techniques, including machine learning and predictive modeling.

In essence, data analytics is a subset of data science. Data science builds upon data analytics by incorporating more advanced

techniques and focusing on prediction and discovery.

1.12 How Big Is Big Data?

The amount of data produced worldwide is enormous, and its growth is accelerating. **Big data** applications deal with massive amounts of data. The following table presents some common big-data byte measures:

Name	How big is (approximately)
kilobyte	10^3 bytes (1024 bytes exactly)
megabyte	10^6 bytes
gigabyte	10^9 bytes
terabyte	10^{12} bytes
petabyte	10^{15} bytes
exabyte	10^{18} bytes
zettabyte	10^{21} bytes

For computer scientists and data scientists, understanding how to obtain, store, process and extract value from data is as crucial as writing effective programs, and possibly even more so with the booming interest in generative AI. In 2025, it's expected that we'll create 182 zettabytes of data—equivalent to approximately 182 sextillion bytes (182,000 exabytes; one exabyte is 10^{18} bytes) per day—and this is expected to more than double by 2028.[77] The Internet is responsible for much of this data generation. Consider the following real-world examples of various popular data measures.

77. Statista, "Volume of Data/Information Created, Captured, Copied, and Consumed Worldwide from 2010 to 2023, with Forecasts from 2024 to 2028 (in Zettabytes)." Accessed February 18, 2025. https://www.statista.com/statistics/871513/worldwide-data-created/.

Megabytes

One megabyte (MB) is about one million (exactly 2^{20}) bytes. High-quality MP3-format audio files require from 1 to 2.4 MB per minute.[78] JPEG-format photos taken on a digital camera can require 3 to 7 MB per photo. On one of our iPhones, the Camera

settings app reports that 1080p video at 30 frames-per-second (FPS) requires 65 MB/minute, and 4K video at 60 FPS requires 440 MB/minute.

78. “Audio File Size Calculations.” Accessed February 18, 2025. https://www.audiomountain.com/tech/audio-file-size.html.

Gigabytes

One gigabyte (GB) is 1024 megabytes (2^{30} bytes). A DVD can store up to 8.5 GB[79], which could hold 141 hours of MP3 audio, hundreds of photos from a 16-megapixel camera, 7.7 minutes of 1080p video at 30 frames-per-second (FPS), or 2.33 minutes of 4K video at 60 FPS. The current highest-capacity Ultra HD Blu-ray discs can store up to 100 GB of video.[80] Streaming a highly compressed 4K movie can use between 7 and 10 GB per hour.

79. “DVD.” Wikipedia. Wikimedia Foundation. Accessed February 18, 2025. https://en.wikipedia.org/wiki/DVD.

80. “Ultra HD Blu-ray.” Wikipedia. Wikimedia Foundation. Accessed February 18, 2025. https://en.wikipedia.org/wiki/Ultra_HD_Blu-ray.

Terabytes

One terabyte (TB) is 1024 gigabytes (2^{40} bytes). Recent disk drives for desktop computers come in sizes up to 36 TB,[81] which could hold 50.4 years of MP3 audio, hundreds of thousands of photos from a 16-megapixel camera, 131.4 hours of 1080p video at 30 frames-per-second (FPS), or 39.7 hours of 4K video at 30 FPS.

81. “Hard disk drive.” Wikipedia. Wikimedia Foundation. Last modified February 10, 2025. Accessed February 18, 2025. https://en.wikipedia.org/wiki/Hard_disk_drive.

Petabytes, Exabytes and Zettabytes

In 2021, there were almost five billion people online creating about 2.5 quintillion bytes of data each day[82]—that’s 2500 petabytes (each petabyte is 1024 terabytes) or 2.5 exabytes (each exabyte is 1024 petabytes). In 2025, there are expected to be 41.6 billion active IoT devices,[83] and worldwide data produced is expected to reach 163 zettabytes—about 360 MB per minute for every person on the planet. That’s about 51.6 petabytes of new data per second, 3100 petabytes per minute, 186 exabytes per hour, or 446 exabytes (almost half a zettabyte) per day. That’s billions of hours of 4K video or trillions of photos every day!

82. Aditya Rayaprolu, “How Much Data Is Created Every Day in 2024?” Updated March 12, 2024. Accessed February 18, 2025. https://techjury.net/blog/how-much-data-is-created-every-day.

83. Dell Technologies, “Internet of Things and Data Placement,” Edge to Core and the Internet of Things, Accessed February 18, 2025, https://infohub.delltechnologies.com/en-us/l/edge-to-core-and-the-internet-of-things-2/internet-of-things-and-data-placement/.

Additional Big-Data Stats

For a real-time sense of big data, check out https://www.worldometers.info/, with various statistics, including that day’s numbers of Google searches, Tweets (now called posts on X) and e-mails sent. Domo, Inc.’s infographic called “Data Never Sleeps”[84] shows interesting statistics regarding how much data was generated every minute in 2022 (these numbers have increased since), including:

- 66,000 photos posted on Instagram.
- 1.7 million Facebook posts.
- 2.43 million snaps on SnapChat.
- 1 million hours of streaming video viewed.
- 104,600 hours spent in Zoom meetings.

84. “Data Never Sleeps.” Accessed February 18, 2025. https://www.domo.com/data-never-sleeps.

Processing the World’s Data Requires Lots of Electricity

A 2015 article stated that energy use for processing the world’s data was growing at 20% per year and consuming approximately 3–5% of the world’s power.[85] A recent article from MIT’s Sloan School of Management indicates that data centers might consume 21% of the world’s power by 2030.[86]

85. “‘Tsunami of data’ could consume one fifth of global electricity by 2025.” December 11, 2017. Accessed February 18, 2025. https://www.theguardian.com/environment/2017/dec/11/tsunami-of-data-could-consume-fifth-global-electricity-by-2025.

86. Beth Stackpole, “AI Has High Data Center Energy Costs — but There Are Solutions,” MIT Sloan School of Management, January 7, 2025. Accessed March 5, 2025. https://mitsloan.mit.edu/ideas-made-to-matter/ai-has-high-data-center-energy-costs-there-are-solutions.

Another enormous electricity consumer is the blockchain-based cryptocurrency Bit-coin. Processing just one Bitcoin transaction uses approximately the same amount of energy as powering the average American home for a week. The energy use comes from the process Bitcoin "miners" use to prove that transaction data is valid.[87] This energy-consumption level is unsustainable, especially given the huge interest in blockchain-based applications, even beyond cryptocurrencies. The blockchain community is working on fixes.[88] Generative AI is also a big energy consumer. For example, ChatGPT's daily energy consumption is the equivalent of powering 180,000 U.S. homes.[89]

87. "One Bitcoin Transaction Consumes As Much Energy As Your House Uses in a Week." November 1, 2017. Accessed February 18, 2025. https://motherboard.vice.com/en_us/article/ywbbpm/bitcoin-mining-electricity-consumption-ethereum-energy-climate-change.

88. Samuel Huestis, "Cryptocurrency's Energy Consumption Problem," Rocky Mountain Institute, January 30, 2023. Accessed February 18, 2025. https://rmi.org/cryptocurrencys-energy-consumption-problem/.

89. Gordon, Cindy. "ChatGPT and Generative AI Innovations Are Creating Sustainability Havoc." Forbes, March 12, 2024. Accessed February 18, 2025. https://www.forbes.com/sites/cindygordon/2024/03/12/chatgpt-and-generative-ai-innovations-are-creating-sustainability-havoc/.

1.12.1 Big-Data Analytics

The term "data analysis" was coined in the 1960s, though people have been analyzing data using statistics for thousands of years, going back to the ancient Egyptians.[90] The term "big data" was coined around 1987.[91]

90. "A Brief History of Data Analysis," December 7, 2021. Accessed February 18, 2025. https://www.flydata.com/blog/a-brief-history-of-data-analysis/.

91. Diebold, Francis. (2012). On the Origin(s) and Development of the Term "Big Data". SSRN Electronic Journal. 10.2139/ssrn.2152421. https://www.researchgate.net/publication/255967292_On_the_Origins_and_Development_of_the_Term_.

Consider four of the "V's of big data":[92,93]

- Volume—the data the world is producing is growing exponentially.
- Velocity—the speed at which data is produced, moves

through organizations and changes are all growing quickly.[94][95]

- Variety—today's data includes characters, digits, punctuation and special characters, as well as images, audios, videos and data from the rapidly growing number of Internet of Things sensors in our homes, businesses, vehicles, cities and more.
- Veracity—the validity of the data—is it complete and accurate? Can we trust that data when making crucial decisions? Is it real?

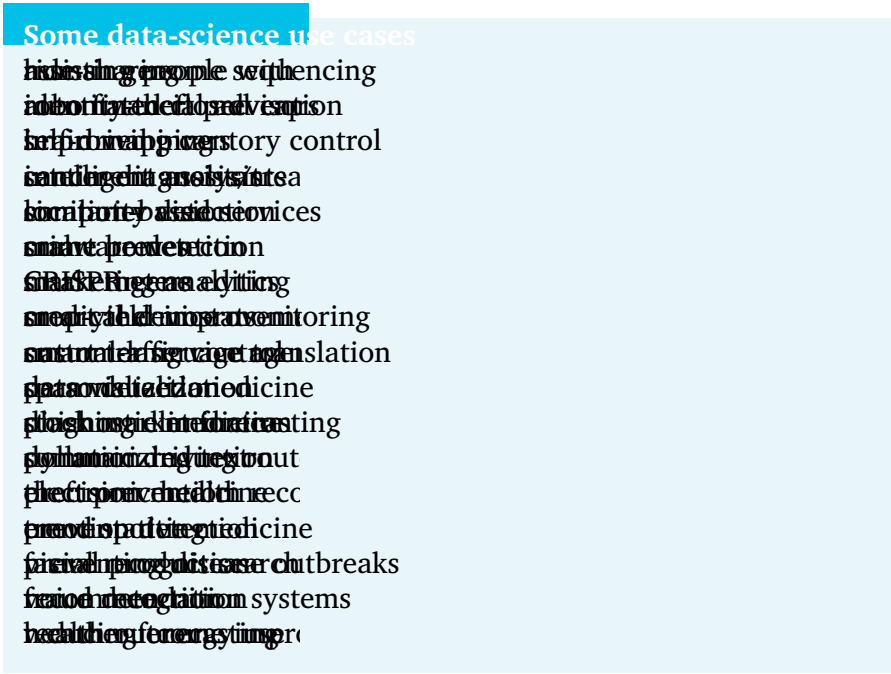
92. "The Four V's of Big Data – What is big data?" December 10, 2021. Accessed February 18, 2025. https://www.analyticsinsight.net/the-four-vs-of-big-data-what-is-big-data/.
93. There are lots of articles and papers that add many other "V-words" to this list.
94. David Gewitz. "Volume, velocity, and variety: Understanding the three V's of Big Data." March 21, 2018. Accessed February 18, 2025. https://www.zdnet.com/article/volume-velocity-and-variety-understanding-the-three-vs-of-big-data/.
95. Ben Lutkevich, "3Vs (volume, variety and velocity)." Accessed February 18, 2025. https://whatis.techtarget.com/definition/3Vs.

Digital data storage has become so vast in capacity, and so cheap and small, that we can now conveniently and economically retain all the digital data we're creating. That's truly big data.

1.12.2 Data Science and Big Data Are Making a Difference: Use Cases

The data-science field is growing rapidly because it produces results that make a difference in our personal and business lives. The following table enumerates many data-science and big-data use cases. Big-data analytics has resulted in improved profits, better customer relations, and even sports teams worldwide winning more games and championships.[96][97][98] In broadcasts of NFL professional football in the United States, the announcers often mention that the coaches are using analytics for fourth-down decision-making.

96. Sawchik, T., *Big Data Baseball: Math, Miracles, and the End of a 20-Year Losing Streak* (New York: Flat Iron Books, 2015).
97. Ayres, I., *Super Crunchers* (Bantam Books, 2007), pp. 7–10.
98. Lewis, M., *Moneyball: The Art of Winning an Unfair Game* (W. W. Norton & Company, 2004).



1.13 AI—at the Intersection of Computer Science and Data Science

When a baby first opens its eyes, does it "see" its parent's faces? Does it understand any notion of a face or even a simple shape? Babies must "learn" the world around them. That's what artificial intelligence (AI) technology called machine learning does. It observes massive amounts of data and learns from it. AI technologies are being used to play games, implement computer-vision applications, enable self-driving cars, enable robots to learn to perform new tasks, diagnose medical conditions, translate speech to other languages in near real-time, create chatbots that can respond smoothly and accurately to arbitrary questions after being trained on massive text databases called **large language models**, and much more. Who'd have guessed just a few years ago that artificially intelligent self-driving cars would be allowed on our roads—or even become common?

1.13.1 Artificial Intelligence (AI)

Two definitions of **artificial intelligence (AI)**[99,100] are:

- "a branch of computer science dealing with the simulation of

intelligent behavior in computers" and

- "the capability of a machine to imitate intelligent human behavior."

99. "artificial intelligence." Merriam-Webster.com. Accessed February 18, 2025. https://www.merriam-webster.com/dictionary/artificial%20intelligence.

100. "Artificial intelligence." Wikipedia. Wikimedia Foundation. Accessed February 18, 2025. https://en.wikipedia.org/wiki/Artificial_intelligence.

The term AI was coined by John McCarthy in 1956. AI-based systems now permeate our daily lives. Examples include computer vision, speech recognition, natural language translation, recommender systems (such as Amazon recommending products you might like or Netflix recommending shows and movies you might like), automated game-playing systems, self-driving cars, and more recent remarkable tools like

- chatbots like OpenAI's ChatGPT, Google's Gemini, Anthropic's Claude and Perplexity that converse almost indistinguishably from conversing with humans,
- image generators like OpenAI's Dall-E (now integrated into ChatGPT), Google's Imagen (integrated into Gemini), Midjourney, Stable Diffusion, Adobe's Firefly and many more that create original images from text descriptions, and
- video generators like OpenAI's Sora, Adobe's Firefly Video, Google's Veo 2, Meta's MovieGen and Runway's Gen-3 Alpha that create original video from text descriptions.

For many decades, AI has been a field with problems and no solutions. That's because once a problem is solved, people say, "Well, that's not intelligence; it's just a computer program that tells the computer exactly what to do." However, with AI techniques like machine learning (Section 1.13.4) and deep learning (Section 1.13.5), we're not pre-programming solutions to specific problems. Instead, we're having computers solve problems by learning from data—typically, lots of it. Many of the most interesting and challenging problems are being pursued with deep learning.

1.13.2 Artificial General Intelligence (AGI)

The ultimate goal of AI is **artificial general intelligence (AGI)**[101]

that, like a human, can understand, learn and apply knowledge across a wide range of tasks without needing extra training—basically, an AI that can "carry out any task that humans are capable of."[102] Experts disagree on when or if artificial general intelligence might become a reality—from 2025[103] to the 2030s[104] to 2300 or never.[105]

101. "Artificial general intelligence." Wikipedia. Wikimedia Foundation. Accessed February 18, 2025. https://en.wikipedia.org/wiki/Artificial_general_intelligence.

102. "Technological singularity." Wikipedia. Wikimedia Foundation. Accessed February 18, 2025. https://en.wikipedia.org/wiki/Technological_singularity.

103. Sam Altman, "Reflections," January 5, 2025. Accessed February 18, 2025. https://blog.samaltman.com/reflections.

104. Lutz Finger, "Sam Altman Says, 'We Know How to Build AGI' — Not So Fast," Forbes, January 7, 2025. Accessed February 18, 2025. https://www.forbes.com/sites/lutzfinger/2025/01/07/sam-altman---agi--ai-workforce-in-2025-the-battle-of-tech-giants/.

105. McKinsey & Company, "What Is Artificial General Intelligence (AGI)?," March 21, 2024. Accessed February 18, 2025. https://www.mckinsey.com/featured-insights/mckinsey-explainers/what-is-artificial-general-intelligence-agi.

1.13.3 Artificial Intelligence Milestones

Several AI milestones have captured people's attention and imagination, made the general public start thinking that AI is real and made businesses think about commercializing it:

- In a 1997 match between **IBM's DeepBlue** computer system and chess Grandmaster Gary Kasparov, DeepBlue became the first computer to beat a reigning world chess champion under tournament conditions.[106] IBM loaded DeepBlue with hundreds of thousands of grandmaster chess games. DeepBlue could use brute force to evaluate up to 200 million moves per second![107] This is big data at work. IBM received the $100,000 Carnegie Mellon University Fredkin Prize, offered in 1980 to the creators of the first computer to beat a world chess champion.[108]

  106. "Deep Blue versus Garry Kasparov." Wikipedia. Wikimedia Foundation. Accessed February 18, 2025. https://en.wikipedia.org/wiki/Deep_Blue_versus_Garry_Kasparov.

  107. "Deep Blue (chess computer)." Wikipedia. Wikimedia

Foundation. Accessed February 18, 2025. https://en.wikipedia.org/wiki/Deep_Blue_(chess_computer).

108. "IBM Deep Blue Team Gets $100,000 Prize." Accessed February 18, 2025. https://articles.latimes.com/1997/jul/30/news/mn-17696.

- In 2011, **IBM's Watson** beat the two best human Jeopardy! players in a $1 million match. Watson simultaneously used hundreds of language-analysis techniques to locate correct answers in 200 million pages of content (including all of Wikipedia), requiring four terabytes of disk storage.[109,110] Watson was trained with AI machine-learning and reinforcement-learning techniques.[111]

109. Jo Best, "IBM Watson: The inside story of how the Jeopardy-winning supercomputer was born, and what it wants to do next." Accessed February 18, 2025. https://www.techrepublic.com/article/ibm-watson-the-inside-story-of-how-the-jeopardy-winning-supercomputer-was-born-and-what-it-wants-to-do-next/.

110. "Watson (computer)." Wikipedia. Wikimedia Foundation. Accessed February 18, 2025. https://en.wikipedia.org/wiki/Watson_(computer).

111. Ferrucci, D., Brown, E., Chu-Carroll, J., Fan, J., Gondek, D., Kalyanpur, A. A., Lally, A., Murdock, J. W., Nyberg, E., Prager, J., Schlaefer, N., & Welty, C. (2010). "Building Watson: An Overview of the DeepQA Project." AI Magazine, 31(3), 59-79. https://doi.org/10.1609/aimag.v31i3.2303.

- Go—a board game created in China thousands of years ago[112]—is widely considered one of the most complex games ever invented, with 10^{170} possible board configurations.[113] To understand how large that number is, it's believed there are (only) approximately 10^{80} atoms in the known universe![114] In 2015, **AlphaGo**—created by Google's DeepMind group—used the AI deep learning technique with two neural networks to beat the European Go champion Fan Hui. Go is considered to be a far more complex game than chess.

112. *Encyclopedia Britannica*, "Go," Last modified February 7, 2025. Accessed February 18, 2025. https://www.britannica.com/topic/go-game.

113. Nsikan Akpan, "Google artificial intelligence beats champion at world's most complicated board game." January 27, 2016. Accessed February 18, 2025. https://www.pbs.org/newshour/science/google-artificial-intelligence-beats-champion-at-worlds-most-complicated-board-game.

114. Paul Sutter, "How Many Atoms Are in the Observable Universe?" Live Science, July 11, 2021. Accessed February 18, 2025. https://www.livescience.com/how-many-atoms-in-universe.html.

- In 2017, Google generalized its AlphaGo AI to create **AlphaZero**—a game-playing AI that teaches itself to play other games. In December 2017, AlphaZero learned the rules and taught itself to play chess in less than four hours using the AI technique of reinforcement learning. It then beat the world champion chess program, Stockfish 8, in a 100-game match—winning or drawing every game. After training itself in Go for just eight hours, AlphaZero beat its AlphaGo predecessor in 60 of 100 games.115

115. Samuel Gibbs, "AlphaZero AI beats champion chess program after teaching itself in four hours." December 7, 2017. Accessed February 18, 2025. https://www.theguardian.com/technology/2017/dec/07/alphazero-google-deepmind-ai-beats-champion-program-teaching-itself-to-play-four-hours.

- In 2023, Google announced AlphaDev, which is built on top of AlphaZero. AlphaDev is designed to discover enhanced computer science algorithms, specifically focusing on sorting algorithms. "AlphaDev uncovered new sorting algorithms that led to improvements in the LLVM libc++ sorting library that were up to 70% faster for shorter sequences and about 1.7% faster for sequences exceeding 250,000 elements."116

116. Daniel J. Mankowitz and Andrea Michi, "AlphaDev Discovers Faster Sorting Algorithms," Deep-Mind, June 7, 2023. Accessed February 18, 2025. https://deepmind.google/discover/blog/alphadev-discovers-faster-sorting-algorithms/.

1.13.4 Machine Learning

Machine learning is "a subfield of artificial intelligence that gives computers the ability to learn without explicitly being programmed."117 Machine learning typically uses predefined algorithms to build models that learn from training data. Those models can then be used to make predictions.118 For example, given labeled training data that contains legitimate e-mails and junk (also called spam) e-mails, a machine-learning model can learn from the training data, then be used to classify e-mails it has never "seen" as junk or not junk. E-mail apps like Microsoft's Outlook and Google's Gmail use techniques like this to move junk emails out of

your inbox.

117. Sara Brown “Machine learning, explained.” April 21, 2021. Accessed February 18, 2025. https://mitsloan.mit.edu/ideas-made-to-matter/machine-learning-explained.

118. “Machine learning.” Wikipedia. Wikimedia Foundation. Accessed February 18, 2025. https://en.wikipedia.org/wiki/Machine_learning.

1.13.5 Deep Learning

Deep learning is a subset of machine learning that involves building artificial neural networks, which “attempt to simulate the network of neurons that make up a human brain so that the computer will be able to learn things and make decisions in a human-like manner.”[119] The term “deep” comes from the number of software “layers” in an artificial neural network—the more layers, the deeper the network. The network learns more from its training data as it proceeds through the network layers.[120] The availability of big data, significant processor power, faster Internet speeds and advancements in parallel computing hardware and software are making it possible for more organizations and individuals to pursue resource-intensive deep-learning solutions that are producing impressive results in computer vision, speech recognition, speech synthesis and newer generative AI applications (Section 1.14).

119. Jason Brownlee, “What is deep learning?” November 16, 2023. Accessed February 18, 2025. https://machinelearningmastery.com/what-is-deep-learning/.

120. “Deep learning.” Wikipedia. Wikimedia Foundation. Accessed February 18, 2025. https://en.wikipedia.org/wiki/Deep_learning.

1.13.6 Reinforcement Learning

Reinforcement learning[121,122] is a subset of machine learning in which algorithms learn from their environment, similar to how humans learn—for example, a video game enthusiast learning a new game or a baby learning to walk or recognize its parents. The algorithm implements an agent that learns by trying to perform a task, receiving feedback about success or failure, making adjustments then trying again. The goal is to maximize the reward. The agent receives a positive reward for doing something right and a negative reward (a punishment) for doing something wrong. The

agent uses this information to determine the next action to perform and must try to maximize the reward.

121. "Reinforcement learning." Wikipedia. Wikimedia Foundation. Accessed February 18, 2025. https://en.wikipedia.org/wiki/Reinforcement_learning.

122. Abid Ali Awan, "Reinforcement Learning for Newbies," KDnuggets, May 16, 2022. Accessed February 18, 2025. https://www.kdnuggets.com/2022/05/reinforcement-learning-newbies.html.

1.14 Generative AI

Generative AI (genAI) describes algorithms that can generate new content—including text, images, audio and video—based on prompts provided by a user as input.[123][124][125]

123. "What Is Generative AI?" McKinsey & Company, April 2, 2024. Accessed February 18, 2025. https://www.mckinsey.com/featured-insights/mckinsey-explainers/what-is-generative-ai.

124. "Generative artificial intelligence." Wikipedia. Wikimedia Foundation. Accessed February 18, 2025. https://en.wikipedia.org/wiki/Generative_artificial_intelligence.

125. Hannah Wren, "A Beginner's Guide to Generative AI for Business," Zendesk, October 22, 2024. Accessed February 18, 2025. https://www.zendesk.com/blog/generative-ai-guide/.

GenAI Chatbots

OpenAI's **ChatGPT**, Google's **Gemini**, Anthropic's **Claude** and **Perplexity** are AI-based chatbots you can converse with using natural language.[126] For example, you could prompt it with text like:

- "Tell me about the history of the Java programming language."
- "Who coined the term Metaverse?"
- "Summarize the article at" followed by a link to the article.

126. "Introducing ChatGPT." November 30, 2022. Accessed February 18, 2025. https://openai.com/blog/chatgpt.

Each chatbot then responds with answers that look and feel as if

they were generated by a human.

Multimodal Chatbots

When genAIs were first introduced, they typically received a text prompt and generated a specific type of output, such as text or an image. Many genAI chatbots are now multi-modal, enabling them to understand text, images, audio and even video inputs and use them to generate outputs combining text, images, audio and video. For example, you can prompt genAIs with a speech audio file and instructions asking them to transcribe the speech to text, display the transcription, analyze its sentiment (positive, negative or neutral), translate it to another spoken language and synthesize speech in that other language—and even create images and videos from the generated text! These capabilities were out of reach even to the most sophisticated developers just a few years ago. Today, they're available to the general public.

Turing Test

When you converse with ChatGPT and similar tools, it may seem like you're chatting with another person. In 1950, Alan Turing proposed the **imitation game**—now known as the **Turing test**—as a way to determine whether a computer exhibits truly intelligent behavior.[127] In it, three computer terminals would be hidden from one another, with two people operating terminals via text-based inputs and the third being computer-operated. One person would ask text-based questions that would be sent to the other two terminals. The second person and the computer would send text responses. If the person asking the questions could not determine whether the answers came from the other person or the computer, then the computer would be considered to have exhibited intelligent behavior.

127. "Turing test." Wikipedia. Wikimedia Foundation. Accessed February 18, 2025. https://en.wikipedia.org/wiki/Turing_test.

Things GenAIs Can Do

We prompted ChatGPT, asking for a list of all the types of things generative AIs can generate.[128] We then asked it to summarize its lengthy response:

- **Text**—Articles, essays, stories, poems, lyrics, code, summaries, marketing copy, academic writing, emails, and

instructional guides.

- **Visual/images**—Digital illustrations, concept art, character design, photo-realistic images, logos, branding elements, infographics and AI-enhanced graphics.
- **Audio**—AI-composed music, sound effects, ambient audio, text-to-speech, narration, voice cloning and synthesized speech.
- **Video**—AI-generated animations, explainer videos, video synthesis from text, deepfake videos and virtual characters.
- **3D**—3D models for games, simulations, virtual reality, digital sculptures, animations and special effects.
- **Code and software**—Automated code generation, programming assistance, chat-bots, virtual assistants, automation scripts and machine-learning algorithms.
- **Interactive**—Chatbot conversations, dynamic storytelling, personalized quizzes, surveys, interactive website and educational tools.
- **Data-related**—AI-generated data analysis, reports, visualizations, graphs, charts, tables and synthetic datasets.
- **Educational**—Study guides, lesson plans, quizzes, textbook sections, tutorials and interactive learning exercises.
- **Multimodal**—AI-generated videos with narration and graphics, interactive simulations, and virtual environments combining text, visuals and user interaction.
- **Art and creative**—Digital comics, graphic novels, mixed media art, style-transfer artwork, custom merchandise designs and AI-generated screenplays.
- **Miscellaneous**—AI-generated recipes, fitness plans, daily affirmations, product names, slogans, personalized recommendations and legal document templates.

128. *ChatGPT* response to "Give me a list of all the types of things generative AIs can generate." October 26, 2024. https://chatgpt.com. We edited the response.

These capabilities make content creation accessible to everyone.

Large Language Models (LLMs)

A **large language model (LLM)**[129,130,131] is an AI that understands

and can generate natural language. The term “large” comes from the fact that LLMs are implemented as artificial neural networks (or neural nets) with billions, and even trillions, of parameters. LLMs are trained on massive amounts of text—articles, books, the Internet, code and more. As neural nets learn, they tune parameters, attempting to produce better results. The parameters neural nets learn help LLMs understand and generate language patterns.

129. “Large Language Model.” Wikipedia. Wikimedia Foundation. Accessed February 18, 2025. https://en.wikipedia.org/wiki/Large_language_model.

130. Sean Michael Kerner, “Large Language Model (LLM),” TechTarget. Accessed February 18, 2025. https://www.techtarget.com/whatis/definition/large-language-model-LLM.

131. Adrian Tam, “What Are Large Language Models?” Machine Learning Mastery, July 20, 2023. Accessed February 18, 2025. https://machinelearningmastery.com/what-are-large-language-models/.

LLMs can generate text that’s indistinguishable from human writing. Some continue to learn and adapt, improving their effectiveness over time. They’re great at natural language tasks like translation, classification, summarization, answering questions, sentiment analysis, creative writing, and more. They’re also great at generating code, explaining code, finding errors in code, refactoring code and more.

Transformers

A **transformer**[132][133][134] is a neural network that processes sequential data, like text. It can understand and recognize relationships between even distant words in a sentence. Transformers also recognize the relative importance of words. Many LLMs use **generative pre-trained transformers (GPTs)**[135] that are trained on massive amounts of text using several artificial intelligence techniques.[136] There are other techniques, like Google’s **Bidirectional Encoder Representations from Transformers (BERT)**, which is geared to natural language understanding (NLU; introduced in Section 1.14).[137]

132. “Transformer (Machine Learning Model).” Wikipedia. Wikimedia Foundation. Accessed February 18, 2025. https://en.wikipedia.org/wiki/Transformer_(machine_learning_model).

133. Rick Merritt, “What Is a Transformer Model?” NVIDIA Blog, March 15, 2022. Accessed February 18, 2025. https://blogs.nvidia.com/blog/what-is-a-transformer-model/.

134. Aveek Goswami, “Understanding Transformers,” Towards Data

Science, Jun 27, 2024. Accessed February 18, 2025. https://medium.com/towards-data-science/understanding-transformers-3344d16c8c36.

135. "Large language model." Wikipedia. Wikimedia Foundation. Accessed February 18, 2025. https://en.wikipedia.org/wiki/Large_language_model.

136. "ChatGPT." Wikipedia. Wikimedia Foundation. Accessed February 18, 2025. https://en.wikipedia.org/wiki/ChatGPT.

137. "BERT (Language Model)." Wikipedia. Wikimedia Foundation. Accessed February 18, 2025. https://en.wikipedia.org/wiki/BERT_(language_model).

Self-Attention

Self-attention[138,139] is a key aspect of transformers, which are used in natural language processing. When trying to understand a word in a sentence, the context provided by surrounding words plays a significant role in determining its meaning. For example:

- The word "good" typically is positive, but when preceded by "not," the context becomes negative.
- The word "bank" could refer to a financial institution or the side of a river, depending on whether it's paired with words like "loan" or "river."[140] Instructors like to use books that are accompanied by a "test bank," which helps them create quizzes.
- Similarly, a phrase like "break the ice" shifts from its literal interpretation to a metaphorical one when used in contexts related to social interactions.[140]

138. Ashish Vaswani et al, "Attention Is All You Need," arXiv preprint arXiv:1706.03762 (2017). Accessed February 18, 2025. https://arxiv.org/abs/1706.03762.

139. "Attention (Machine Learning)." Wikipedia. Wikimedia Foundation. Accessed February 18, 2025. https://en.wikipedia.org/wiki/Attention_(machine_learning).

140. *ChatGPT* response to "Provide examples in which the context provided by surrounding words plays a significant role in determining a word's meaning in a sentence." October 26, 2024. https://chatgpt.com. We edited the response.

Transformers leverage self-attention to analyze each word in relation to the others around it. Since not all words contribute equally to the meaning of a sentence, self-attention assigns different

weights to words, focusing more on the most relevant ones. Unlike humans, transformers can process and analyze vast numbers of words simultaneously, making connections between them to better grasp their overall meaning.

Tokens, Context and Context Windows

A **token**[141] is the unit of processing in language models. Tokens can be words, parts of words or punctuation. Tokenization converts tokens to numerical values models can process. A token's **context** is a sequence of tokens that comes before (and sometimes after) a word or phrase. Understanding the context helps models generate more accurate responses. A **context window**[142] is the maximum number of tokens (input and output combined) a model can consider at once. The larger the context window is, the more context there is to your conversation with a genAI, the more relevant and coherent its responses will be, and the greater the cost to generate a useful response. This is why genAIs' free tiers often have smaller context windows or allow fewer queries in a given time period.

141. Microsoft. "Understanding Tokens." Microsoft Learn, May 20, 2024. Accessed February 18, 2025. https://learn.microsoft.com/en-us/dotnet/ai/conceptual/understanding-tokens.

142. IBM Research. "Why Larger LLM Context Windows Are All the Rage." IBM Research Blog, July 24, 2024. Accessed February 18, 2025. https://research.ibm.com/blog/larger-context-window.

GenAI chatbot context window sizes vary tremendously. For the four we use extensively—OpenAI's ChatGPT, Google's Gemini, Anthropic's Claude and Perplexity—the context window sizes range from a low of 8,000 tokens for ChatGPT's free version[143] to a high of 1,000,000 tokens for Gemini's paid version.[144]

143. *ChatGPT* response to "What's your context window size?" February 15, 2025. https://chatgpt.com.

144. "Gemini Advanced." Accessed February 18, 2025. https://gemini.google/advanced.

Prompts and Prompt Engineering

Prompt engineering is the process of designing prompts to obtain the best responses. Outside Chapter 19, many of the book's sections end with Generative AI exercises in which we focus on prompt engineering to help you interact effectively with genAI tools. Think of this as learning a new and different type of programming

language. With prompt engineering, you craft precise instructions or queries to guide genAIs in producing the required responses.

When you prompt genAIs, be sure to:

- **Include "Java" in your prompt** when you ask genAIs to write Java code or answer questions about Java features. Also, **ask them to write Java code using the latest Java idioms** based on the Java version you're using.
- **Keep track of your prompts and the quality of their results.** This is crucial for learning what works and what doesn't. This will also help you identify the nuances of different genAI platforms. **You'll build a repertoire of valuable prompts and anticipated results.**
- **Experiment with multiple genAIs**—including new ones as they appear—and compare the results to understand their varying capabilities.
- **Experiment with different prompting strategies** to see which ones help you achieve your desired outcomes. Different programming challenges can require different prompting strategies.
- **Tune your prompt and try again if the genAIs' results are not satisfactory.** Better prompts produce better results. Some genAIs dig deeper into the subject matter.

When prompting some genAIs, pressing *Enter* (or *Return*) submits your prompt immediately. For others, you must explicitly press (or select) a button to submit the prompt. **To move the cursor to a new line without submitting the prompt**, use

Shift + *Enter*

or

Shift + *Return*

When we ask you to provide Java code to the genAIs as part of a prompt in Chapters 2 and higher, we recommend that you copy and paste the code from Java source-code files. **Most genAIs also allow you to upload one or more files**, but this is sometimes a feature provided only for paid subscribers.

It takes lots of experience to learn how to create great prompts. Various online guides to help you build better prompts, and you can

even ask the genAIs themselves to tune your prompts. **OpenAI provides a nice prompt engineering guide** at

Click here to view code image

```
https://platform.openai.com/docs/guides/prompt-engineering
```

and an **extensive set of sample prompts** at

Click here to view code image

```
https://platform.openai.com/docs/examples
```

Generative AI Anecdotes

During November and December of 2023, we would ask ChatGPT to perform tasks for which it previously generated detailed responses. Rather than detailed responses, it was lazy and provided steps telling us how to perform the tasks ourselves. Some researchers proposed the "winter break hypothesis,"145 speculating that ChatGPT learned to do less work at that time of year because people in the west tend to do less around the holiday season.

145. Edwards, Benj. "As ChatGPT gets "lazy," people test "winter break hypothesis" as the cause." December 11, 2023. Accessed October 13, 2024. https://arstechnica.com/information-technology/2023/12/is-chatgpt-becoming-lazier-because-its-december-people-run-tests-to-find-out/.

Though powerful, genAI chatbots sometimes make errors or even "**hallucinate**,"146 confidently presenting incorrect information as facts. As one example of an error, we asked several genAIs, "What do these Java statements output?" and fed them five simple statements that display the following three lines of asterisks:

```
****
*****
******
```

146. "Hallucination (artificial intelligence)." Wikipedia. Wikimedia Foundation. Accessed February 18, 2025. https://en.wikipedia.org/wiki/Hallucination_(artificial_intelligence).

The typical genAI response was:

```
***************
```

or to display one asterisk per line vertically.

Around August 2024, someone discovered that asking ChatGPT's

voice mode, “How many R’s are in strawberry?” yielded the incorrect spoken response, “There are two R’s in strawberry.”[147,148] Interestingly, entering the same question as a text prompt yielded the correct answer, “There are three R’s in strawberry.” On March 30, 2025, we repeated this experiment with ChatGPT’s voice mode, and it once again said, “There are two R’s in strawberry.” We said, “No. There are three R’s in strawberry.” The ChatGPT voice came back and said, “Let’s see: S-T-R-A-W-B-E-R-R-Y. You’re right, there are three R’s!”

147. Kit Eaton, “How Many R’s in Strawberry? This AI Can’t Tell You,” Inc., August 28, 2024. Accessed March 1, 2023. https://www.inc.com/kit-eaton/how-many-rs-in-strawberry-this-ai-cant-tell-you.html.

148. We tried this in March 2025, and ChatGPT voice mode still gives the wrong answer.

One example of hallucinating is that GenAIs have been known to make up court cases. You might research this with the following prompt, “Give me citations for court cases in which judges reprimanded lawyers for citing non-existent court cases generated by AIs.”

Another example of hallucinating comes from our experience in the early days of ChatGPT. We wanted to experiment with its ability to create text-based content. We asked it to write definitions of various computing terms and to provide four references for each—one of which we required to be Wikipedia. In general, it produced nice definitions and gave a direct link to the correct Wikipedia page. Without fail, however, the other three references we requested were completely fabricated—fake names, fake quotes, fake websites, and sometimes real websites but pages that did not exist. Though they are much better now than in their early days, you should always verify the information genAIs provide. Also, data on the Internet can have biases. When genAIs are trained, they might pick up those biases. We discuss these issues in Exercise 1.23 and in a section in Chapter 19.

Natural Language Processing (NLP) and Natural Language Understanding (NLU)[149]

149. *Claude* response to “Please write a paragraph explaining what natural language processing and natural language understanding are and how they’re fundamental to generative AI.” February 18, 2024. https://claude.ai. We edited the response.

Natural language processing (NLP) and **natural language understanding (NLU)** are two key technologies that enable genAIs

to interact with human language. NLP focuses on how computers process and analyze human language—breaking down text into meaningful pieces like words, sentences and relationships between them.[150] Think of NLP as the system that helps computers read and make sense of text, much like how you're currently reading and processing these words.

150. The phrase "red brick factory" illustrates why natural language is such a challenging subject. Is a "red brick factory" a factory that makes red bricks? Is it a red factory that makes bricks of any color? Is it a factory built of red bricks that makes products of any type? In today's music world, it could even be a rock band's name or the name of a game on your smartphone.

NLU helps computers understand the meaning and context of language, not just the individual words. For example, when you ask a genAI, "What's the weather like?" then ask, "Should I bring an umbrella?", the genAI uses NLU to recognize that your second question is related to the weather and that you're asking about rain protection.

NLP and NLU form the foundation for generative AI to understand your prompts and generate human-like responses. Without them, interacting with generative AI would be like trying to converse with someone who knows all the words in a language but doesn't understand how they fit together or what they actually mean.

1.15 Wrap-Up

This chapter introduced terminology and concepts that lay the groundwork for the Java programming capabilities you'll learn throughout the book. We discussed Moore's Law, noting that the steady stream of dramatic improvements it so accurately predicted for decades seems to be ending. As a result, hardware designers have shifted to multi-core architectures, which you'll program with the multithreading techniques we discuss in Chapter 18, Concurrency: Platform Threads to Virtual Threads. We introduced Java and encouraged you to avoid reinventing the wheel when developing new software by using preexisting software components from the Java API and open-source Java libraries.

We briefly reviewed object orientation and the notions of classes, objects, interfaces, inheritance and polymorphism, providing the groundwork for the objects-natural case studies in Chapters 2–7 and 11 and our detailed treatment of object-oriented programming in

Chapters 8–9.

We discussed a typical Java program-development environment in the context of the Java Development Kit (JDK) then used its tools to test-drive a Java program to prepare you to run the book's example programs and original programs you'll write. You can also run our programs with a Java integrated development environment (IDE)—we provide test-drives for several popular IDEs at `https://deitel.com/javafp5`.

We discussed the Internet, the web, the cloud and the Internet of Things (IoT). We introduced the Metaverse and key associated technologies—blockchain, cryptocurrencies, non-fungible tokens (NFTs), Web3, augmented reality (AR), virtual reality (VR) and mixed reality. We considered various software development technologies, including refactoring, design patterns, software development kits (SDKs), version control systems (VCS), agile software development, test-driven development (TDD), microservices and containerization.

We introduced big data analytics, then discussed big data and various data sizes, from megabytes to exabytes. Finally, we introduced artificial intelligence (AI)—an intersection between computer science and data science—presenting various AI milestones and brief overviews of several AI technologies, including artificial general intelligence (AGI), machine learning, deep learning, reinforcement learning and generative AI, which you'll use throughout this book to leverage your Java learning experience. In this chapter's end-of-chapter exercises, you'll begin using generative AI to enhance your Java learning experience. In Chapter 2, you'll begin programming in Java.

Special Section: Generative AI Exercises

For each of the following questions, try multiple genAIs to determine which performs best. Always verify the results. Keep in mind that they sometimes make mistakes and even hallucinate, creating unreliable information.

1.1 ***(2010 "Flash Crash")*** An example of the consequences of our dependency on computers was the so-called "flash crash," which occurred on May 6, 2010, when the U.S. stock market fell precipitously in minutes, wiping out over one trillion dollars of investments—then recovering in minutes. Use genAIs to investigate the causes of this crash and discuss the issues it raises.

1.2 ***(Java Compared to Modern Programming Languages)*** Prompt genAIs to list ways in which Java may be considered first among modern programming languages.

1.3 ***(Java's Six-Month Release Cadence)*** Prompt genAIs to explain the rationale, pros and cons of Java's six-month release cadence. Ask what challenges this presents to companies that use Java for business- and mission-critical software development.

1.4 ***(Same Question, Different Answer)*** Prompt genAIs, asking why when you ask them the same question seconds, minutes, hours, days or more later, they often return significantly different responses each time.

1.5 ***(OpenAI ChatGPT Capabilities)*** Prompt OpenAI's ChatGPT

```
https://chatgpt.com
```

for a tutorial on its multimodal capabilities for working with text, images, audio and video (if any). Ask it to suggest the kinds of multimodal apps you can build.

1.6 ***(Google Gemini Capabilities)*** Prompt Google's Gemini

```
https://gemini.google.com/app
```

for a tutorial on its multimodal capabilities for working with text, images, audio and video (if any). Ask it to suggest the kinds of multimodal apps you can build.

1.7 ***(Anthropic Claude Capabilities)*** Prompt Anthropic's Claude

```
https://claude.ai
```

for a tutorial on its multimodal capabilities for working with text, images, audio and video (if any). Ask it to suggest the kinds of multimodal apps you can build.

1.8 ***(Perplexity Capabilities)*** Prompt Perplexity

```
https://www.perplexity.ai/
```

for a tutorial on its multimodal capabilities for working with text, images, audio and video (if any). Ask it to suggest the kinds of multimodal apps you can build.

1.9 ***(GenAI Job Opportunities)*** Prompt genAIs asking what you should do to prepare for a career in generative AI and what kinds of

new job opportunities generative AI is creating.

1.10 ***(Technical Innovations and Jobs)*** Some people are concerned that genAIs will eliminate jobs. Prompt genAIs for examples of other major technical innovations—such as the Internet, Google Search and others—that eliminated some jobs but also created new ones.

1.11 ***(Verify Generative AI Responses)*** Prompt genAIs, asking why they sometimes hallucinate and to provide best practices on how to detect and deal with this problem.

1.12 ***(Funny Generative AI Anecdotes)*** Prompt genAIs for a list of funny generative AI anecdotes.

1.13 ***(Free Generative AIs)*** Prompt genAIs to list free genAIs or ones that have a free tier and/or a free trial.

1.14 ***(Video Generation)*** Some genAI services with limited free video generation include:

- Synthesia.io (`https://synthesia.io`) lets you create up to three minutes of video per month, using nine avatars and various AI voices in 130+ languages.
- Invideo AI (`https://invideo.io`) lets you create up to 10 minutes of video per week. Based on your text inputs, it can autogenerate scripts, video clips, subtitles, background music, and transitions.
- Pictory AI (`https://pictory.ai`) offers a free trial for three five-minute videos.
- Adobe Firefly (`https://firefly.adobe.com`) offers free monthly credits.

Prompt generative AIs to give you links to demos produced by each service. Try and comment on these demos.

1.15 ***(Free Text-to-Speech Services)*** Two popular services are:

- Google Cloud's Text-to-Speech offers one million free characters per month for their WaveNet (natural-sounding) voices and four million free characters per month for their standard voices.
- Amazon Polly's free tier allows five million characters per month using their standard voices and one million characters per month using their neural (natural-sounding) voices for

the first 12 months.

Prompt generative AIs to give you links to demos produced by each service. Try and comment on these demos.

1.16 ***(Detecting AI Generated Content)*** Prompt generative AIs to find free tools and tools with free tiers or free trials for detecting AI-generated content. This is a difficult problem and the reported results are often wrong. Comment on the ethics of using these tools.

1.17 ***(Ethics)*** Prompt generative AIs to write tutorials on ethics in AI and list the ethical issues of using generative AI.

1.18 ***(Negatives)*** Prompt generative AIs to list AI negatives.

1.19 ***(Regulation)*** Prompt generative AIs to write tutorials on government regulation of AIs. These cans vary by state or country.

1.20 ***(Safety)*** Prompt generative AIs to write tutorials on AI safety.

1.21 ***(IQ Test)*** Prompt generative AIs to list the types of questions that might appear on IQ tests.

1.22 ***(Anecdote: Decisions at the Speed of Light)*** During the United States Civil War, the telegraph was used for near real-time communication.[151] There's an anecdote attributed to Abraham Lincoln in which one of his young lieutenants excitedly tells him that they're wiring the battlefields with telegraph capabilities. The lieutenant tells the president he'll be able to make decisions with the speed of light. The older and wiser Lincoln responds to the lieutenant that he'll also be able to make *wrong* decisions with the speed of light.

151. "Civil War Signals," Accessed June 24, 2024. https://www.nsa.gov/portals/75/documents/resources/everyone/digital-media-center/video-audio/general/civil-war-signals-transcript.pdf.

We asked generative AIs to track down this anecdote. Here's what ChatGPT said: "The quote you are referring to is often attributed to Abraham Lincoln regarding the use of the telegraph during the Civil War. However, it appears to be a paraphrasing rather than a direct quote from Lincoln. The idea is that the rapid communication provided by the telegraph allowed for quicker decision-making, but also meant that wrong decisions could be disseminated just as quickly. Here is the commonly referenced paraphrased idea: 'With the advent of the telegraph, wrong decisions could now be transmitted at the speed of light.' This reflects Lincoln's awareness of how new technology, like the

telegraph, could impact the speed and consequences of decision-making during the Civil War. The telegraph was indeed a critical tool for communication and coordination in that era, and while it had many benefits, it also meant that mistakes could be communicated and acted upon much more quickly than before."[152]

152. OpenAI ChatGPT 4o, Accessed June 24, 2024. https://chatgpt.com.

Generative AI is a relatively new technology with many ramifications that must be carefully considered. One of its advantages over the Lincoln telegraph anecdote is that it offers many possibilities for improving decision quality. Prompt generative AIs for how they can help you improve decision quality.

1.23 ***(Issues with Generative AI: Biases)*** Data on the Internet has lots of biases and that data is being used to train genAI large language models. So, when the models are trained, there is a chance they'll pick up those biases. Prompt genAIs, asking what they do to avoid picking up biases from the training data.

1.24 ***(Issues with Generative AI: Copyright Issues)*** Prompt genAIs, asking how they ensure the images they create are original and do not violate copyrights.

1.25 ***(AI Agents)*** Prompt genAIs, asking what an AI agent is and to give several prominent examples of them.

1.26 ***(Humans in the Loop)*** Prompt genAIs, asking about the importance of keeping humans "in the loop" when using genAI results, especially in business-critical and mission-critical situations.

Intro to Java Programming

Objectives

In this chapter, you'll:

- Write, compile and run Java programs.
- Use input and output statements.
- Declare, initialize and use variables of primitive types.
- Use arithmetic operators.
- Understand the precedence of arithmetic operators.
- Write decision-making statements.
- Use relational and equality operators to form conditions.
- Use generative AI to leverage your Java learning experience.
- Begin appreciating the "objects-natural" learning approach by conveniently creating and using powerful objects of the Java API's `String` class before you learn to code your own custom classes in later chapters.

Outline

2.1 Introduction

2.2 Your First Program in Java: Printing a Line of Text

2.2.1 Compiling the Program

2.2.2 Executing the Program

2.3 Modifying Your First Program

2.4 Displaying Text with `printf`

2.5 Another Program: Adding Integers

2.6 Arithmetic

2.7 Decision Making: Equality and Relational Operators

2.8 Objects-Natural Case Study: Creating and Using Objects of the Java API's `String` Class

2.9 Wrap-Up

Special Section: Generative AI Exercises

2.1 Introduction

This chapter introduces Java programming. We begin with simple programs that output messages on the screen. We then present a program that inputs (that is, reads) two numbers from a user, calculates their sum and displays the result. You'll perform arithmetic calculations and save their results for later use. The last example demonstrates decision-making by comparing two numbers then displays messages showing the comparison results. Throughout the chapter, you'll use generative AI to leverage your Java learning experience. You'll use the JDK's command-line tools to compile and run this chapter's programs. If you prefer to use an integrated development environment (IDE), see our Test-Drives of popular IDEs at

```
https://deitel.com/javafp5
```

The "Objects-Natural" Learning Approach

In your programs throughout this book, you'll create and use many objects of preexisting carefully developed and tested classes that enable you to perform significant tasks with minimal code, even in the book's earliest chapters. These classes typically come from:

- the Java Application Programming Interface (Java API), and
- free third-party libraries created by the massive open-source communities that have developed around key programming languages.

To help you appreciate this programming style even in the book's introductory chapters, you'll conveniently create and use powerful objects of preexisting Java API classes without yet knowing how to create custom Java classes. We call this the objects-natural approach. In this chapter's final example, you'll begin by creating and using `String` objects. Then, in Chapters 8–9, you'll craft valuable custom classes for your own use and for reuse by other programmers.

2.2 Your First Program in Java: Printing a Line of Text

Consider a simple program that displays a line of text. Figure 2.1 shows the program followed by a box that displays its output. We use line numbers for instructional purposes—they're not part of the program. This example illustrates several important features. We'll see that line 7 performs the program's key action—displaying the text `"Welcome to Java Programming!"` on the screen.

Click here to view code image

```
1  // Fig. 2.1: Welcome1.java
2  // Text-printing program.
3
4  public class Welcome1 {
5     // main method begins program execution
6     public static void main(String[] args) {
7        System.out.println("Welcome to Java Programming!");
8     } // end method main
9  } // end class Welcome1
```

```
Welcome to Java Programming!
```

Fig. 2.1 | Text-printing program.

Commenting Your Programs

We insert **comments** to document programs and improve their readability. The compiler ignores comments, so they do not cause the computer to perform any action when you run the program. By convention, we begin every program with a comment indicating the figure number and the program's filename to help you find the file in the book's `examples` folder (discussed in the Before You Begin section). The comment in line 1 indicates that you can find `Welcome1.java` in the `examples/ch02/fig02_01` folder. This comment begins with `//`, indicating it's an **end-of-line comment** that terminates at the end of the line. Such comments need not begin a line; they also can begin in the middle of a line and continue until the end (as in lines 5, 8 and 9). By our convention, line 2 is a comment describing the program's purpose.

Java also has **traditional comments** that you can spread over several lines, as in

Click here to view code image

```
/* This is a traditional comment. It
   can be split over multiple lines */
```

These begin with the delimiter `/*` and end with `*/`. The compiler ignores all text between the delimiters. Some organizations require programmers to start every source-code file with a multiline comment, including the program's purpose, the programmer's name, and the date and time the programmer last modified the program.

Javadoc Comments

A third comment type, called a **Javadoc comment**, is delimited by `/**` and `*/`. The compiler ignores all text between the delimiters. Javadoc comments enable you to embed program documentation directly in your programs and are preferred in industry. The **`javadoc` utility program**[1,2] (part of the JDK) reads Javadoc comments and uses them to prepare program documentation that's viewed in a web browser. We use `//` comments throughout our code to save space.

1. "How to Write Doc Comments for the Javadoc Tool." Accessed May 17, 2024. https://oracle.com/technical-resources/articles/java/javadoc-tool.html.
2. "The `javadoc` Command." Accessed September 21, 2024. https://docs.oracle.com/en/java/javase/23/docs/specs/man/javadoc.html.

Syntax Errors

Forgetting a comment delimiter is a **syntax error**—such errors occur when the compiler encounters code that violates Java's language rules (that is, its syntax). These rules are similar to natural-language grammar rules specifying sentence structure, such as those in English, Spanish, Chinese, Hindi, etc. Syntax errors are also called **compilation errors** because they're detected by the compiler. A program must compile without errors before you can execute it to perform its job.

Using Blank Lines

Blank lines (like line 3), space characters and tabs—known as **whitespace**—can make programs easier to read. The compiler ignores whitespace.

Declaring a Class

Line 4 begins class `Welcome1`'s **class declaration**. Every program consists of at least one class that you declare. The **`class` keyword** introduces a class declaration and is immediately followed by the **class name** (`Welcome1`). **Keywords** are reserved for use by Java. Each is spelled with all lowercase letters. Most classes we declare begin with the **`public` keyword**. You'll learn more about `public` and non-`public` classes in Chapter 8. The following table shows the complete keywords list:

Keywords

```
abstract      assert        boolean       break
byte          case          catch         char
class         const         continue      default
do            double        else          enum
extends       false         final         finally
float         for           goto          if
implements    import        instanceof    int
interface     long          native        new
null          package       private       protected
public        return        short         static
strictfp      super         switch        synchronized
this          throw         throws        transient
true          try           void          volatile
while         _ (underscore)
```

Contextual keywords (reserved for use in certain program contexts)

```
exports       module        open          opens
provides      requires      to            transitive
uses          with
```

Reserved keywords not currently used by the language

```
const         goto
```

Filename for a `public` Class

A `public` class must be placed in a file with a filename of the form *ClassName*`.java`, so `Welcome1` is stored in the file `Welcome1.java`. A compilation error occurs if a `public` class's filename differs from the class name (in spelling or capitalization) followed by `.java`.

Class Names and Identifiers

By convention, every word in a class-name identifier begins with an uppercase letter. For example, the name `DollarAmount` starts its first word (`Dollar`) with an uppercase `D` and its second word (`Amount`) with an uppercase `A`. This naming convention is known as **camel case** because the uppercase letters stand out like a camel's humps.

A class name is an **identifier**—a series of characters consisting of letters, digits, underscores (_) and dollar signs (`$`) that does not

begin with a digit and does not contain spaces. Some valid identifiers are `Welcome1`, `$value`, `_value`, `m_inputField1` and `button7`. The name `7button` is invalid because it begins with a digit, and the name `input field` is invalid because it contains a space. Java is **case sensitive**—uppercase and lowercase letters are distinct—so `value` and `Value` are different valid identifiers.

Class Body

A **left brace** (`{` at the end of line 4) begins the **class declaration body**. A corresponding **right brace** (`}` at line 9) must end each class declaration.

Lines 5–8 are indented. You should indent the entire body of each class declaration one "level" between the braces. This convention emphasizes the class's structure and makes it easier to read. We use three spaces to form an indent level—many programmers prefer two or four spaces. Whatever you choose, use it consistently. IDEs typically indent code for you, using four-space indents by default. The *Tab* key may also be used to indent code. You can configure each IDE to specify the number of spaces inserted when you press *Tab*.

It's a syntax error if braces do not occur in matching pairs. To prevent this error, when you enter an opening left brace (`{`), immediately enter the closing right brace (`}`). Then, put the cursor between the braces and indent to begin entering the body. In most IDEs, when you enter the opening left brace (`{`), the IDE adds the closing brace, moves the cursor between the braces and indents for you.

Declaring a Method

Line 6 is the starting point of every program. The **parentheses** after the identifier `main` indicate that it's a program building block called a method. Class declarations normally contain one or more methods. One method must be called `main` and must be defined as in line 6; otherwise, the program will not execute.[3]

3. At the time of this writing, a preview feature called "instance `main`" simplifies line 6 to `void main() {`.

Methods perform tasks and can return information when they complete their tasks. We'll explain the purpose of keyword `static` in Section 8.2.5. The **`void` keyword** indicates that this method will not return any information. Later, we'll see methods that return information. For now, mimic `main`'s first line in your programs.

Chapter 6 discusses the `String[] args` in parentheses.

The method's left brace (end of line 6) begins the **method body**. A corresponding right brace (line 8) ends it. You should indent each method's body one "level" between the braces, as in line 7. This emphasizes the method's structure and makes it easier to read.

Performing Output with `System.out.println`

Line 7 displays the characters between the double quotation (`"`) marks. The quotation marks are not displayed. Together, the quotation marks and the characters between them are called a **string**, a **character string** or a **string literal**. The compiler does not ignore whitespace characters in strings. Strings in double quotation (`"`) marks may not span multiple lines of code.[4]

4. Later, we'll discuss text blocks for forming multiline strings.

`System.out` is the **standard output object**, which is predefined for you. It allows programs to display text in the **command window** where you execute the program:

- a Command Prompt or Windows PowerShell on Microsoft Windows, or
- a **terminal** or a **shell** in Linux/macOS.

Many programmers call the command window simply the **command line**.

The `System.out.println` **method** displays a line of text. The string in the parentheses in line 7 is the method's **argument**. When `System.out.println` completes its task, it moves the output cursor (the location where the next character will display) to the beginning of the next line. This is similar to pressing the *Enter* key while entering text in an editor—the cursor moves to the beginning of the next line in the document.

The entire line 7, including the **semicolon** (`;`), is called a **statement**. A method typically contains statements that perform its task. Most statements end with a semicolon.

Using End-of-Line Comments on Right Braces for Readability

For now, we include an end-of-line comment after a closing brace that ends a method declaration and after a closing brace that ends a class declaration. For example, line 8 indicates the closing brace of

the `main` method, and line 9 indicates the closing brace of the `Welcome1` class. We'll omit such ending comments after this chapter.

Generative AI

1 Prompt genAIs with the following code, asking what the code displays:

```
System.out.println("*");
System.out.println("*****");
System.out.println("**");
```

Compare the results to the following output:

```
*
*****
**
```

2.2.1 Compiling the Program

You can now compile and execute the program. Here, we'll use the Java Development Kit's command-line tools. We assume the book's examples are in your user account's `Documents` folder in a subfolder named `examples` as discussed in the Before You Begin section.

Change to the Correct Folder

First, open a command window (see Section 1.7) and change to the folder where the program is stored. Operating systems typically use the command `cd` to change folders. On Windows, change to the `fig02_01` folder by entering the following command and pressing the *Enter* (or *Return*) key:

Click here to view code image

```
cd %userprofile%\Documents\examples\ch02\fig02_01
```

On Linux/macOS, use the following command instead:

Click here to view code image

```
cd ~/Documents/examples/ch02/fig02_01
```

Compile the `.java` File

To compile the program, enter the following command and press

Enter:

```
javac Welcome1.java
```

If the program does not contain compilation errors, this command creates `Welcome1`'s **class file** (`Welcome1.class`) containing the platform-independent bytecodes representing the program. When we use the `java` command to execute the program on a given platform, the JVM will translate these bytecodes into instructions understood by the underlying operating system and hardware.

Error Messages

When compiling, if you receive the error message "`class Welcome1 is public, should be declared in a file named Welcome1.java`," either

- your filename does not match the file's `public` class name or
- you misspelled the class name in the compilation command.

Sometimes, it's helpful to "break" a working program to become familiar with the compiler's error messages. Though error messages generally give you an idea of what caused them, they do not always state the exact problems in the code. Try removing a semicolon or brace from the program of Fig. 2.1, then recompiling to produce the error messages generated by the omission.

If the compiler reports a syntax error, it may not be on the line that the error message indicates. First, check the line reported in the error message. If you don't find an error on that line, check several preceding lines. Each compilation error message contains the filename and line number where the error occurred. For example, `Welcome1.java:6` would indicate that an error occurred at line 6 in `Welcome1.java`. The rest of the message would provide information about the error.

Generative AI

1 Break the program in Fig. 2.1 by removing a brace. Then, prompt genAIs with the broken code and ask them to find and correct the error(s). Compare the fixed code to Fig. 2.1.

2.2.2 Executing the Program

Execute the `Welcome1` program by entering the following command and pressing *Enter*:

```
java Welcome1
```

Notice that you do not include the `.class` filename extension when running the program. The JVM launches and loads the `Welcome1.class` file, then calls `Welcome1`'s `main` method. Next, line 7 in `main` displays

```
Welcome to Java Programming!
```

2.3 Modifying Your First Program

In this section, we modify Fig. 2.1's example to print text on one line using multiple statements and to print text on several lines using one statement.

Displaying a Single Line of Text with Multiple Statements

Class `Welcome2` (Fig. 2.2) uses two statements (lines 7–8) to produce the output in Fig. 2.1.

Click here to view code image

```
1  // Fig. 2.2: Welcome2.java
2  // Printing a line of text with multiple statements.
3
4  public class Welcome2 {
5     // main method begins program execution
6     public static void main(String[] args) {
7        System.out.print("Welcome to ");
8        System.out.println("Java Programming!");
9     } // end method main
10 } // end class Welcome2
```

```
Welcome to Java Programming!
```

Fig. 2.2 | Printing a line of text with multiple statements.

The program is similar to Fig. 2.1, so we discuss only the

changes here. Line 2 is an end-of-line comment stating the purpose of the program. Line 4 begins the `Welcome2` class declaration. Lines 7–8 in the `main` method display one line of text. The first statement uses the `System.out.print` method to display a string. Each `print` or `println` statement resumes displaying characters from where the last statement stopped. Unlike `println`, after displaying its argument, `print` does not position the output cursor at the beginning of the next line—the next character the program displays will appear immediately after the last character that `print` displays. So, line 8 positions the first character in its argument (the letter "`J`") immediately after the last character that line 7 displays (the space character before the string's closing double-quote character).

Displaying Multiple Lines of Text with a Single Statement

A single statement can display multiple lines using **newline characters** (`\n`), which position the output cursor at the beginning of the next line. The program in Fig. 2.3 outputs four lines of text. Most of the program is identical to those in Figs. 2.1 and 2.2.

Click here to view code image

```
1  // Fig. 2.3: Welcome3.java
2  // Printing multiple lines of text with a single statement.
3
4  public class Welcome3 {
5     // main method begins program execution
6     public static void main(String[] args) {
7        System.out.println("Welcome\nto\nJava\nProgramming!");
8     } // end method main
9  } // end class Welcome3
```

```
Welcome
to
Java
Programming!
```

Fig. 2.3 | Printing multiple lines of text with a single statement.

Line 7 displays four lines of text. Normally, each character in a string displays as it appears in the double quotes. However, the paired characters `\` and `n` (repeated three times in the statement) do not appear on the screen. The **backslash** (`\`) is an **escape character**, which has special meaning to `System.out`'s `print` and

`println` methods. When a backslash appears in a string, it's combined with the next character to form an **escape sequence**—the `\n` escape sequence represents the newline character. When `\n` is encountered in a string being output, the output cursor moves to the beginning of the next line.

The following table describes several escape sequences—for the complete list, refer to the Java Language Specification, Section 3.10.7, Escape Sequences:[5]

5. The Java Language Specification, Java SE 22 Edition. Section 3.10.7, "Escape Sequences." Accessed May 30, 2024. https://docs.oracle.com/javase/specs/jls/se22/html/jls-3.html#jls-3.10.7.

Escape sequence	Description
`\n`	Newline. Position the screen cursor at the beginning of the next line.
`\t`	Horizontal tab. Move the screen cursor to the next tab stop.
`\\`	Backslash. Print a backslash character.
`\"`	Double quote. Print a double-quote character. For example,

Click here to view code image

```
System.out.println("\"in quotes\"");
```

displays `"in quotes"`.

Generative AI

1 Prompt genAIs to simplify the following code

```
System.out.print("#");
System.out.println("###");
System.out.println("#####");
System.out.print("####");
System.out.println("##");
```

then run the code they return. Does it produce the following output?

```
####
#####
######
```

[*Note:* At the time of this writing, all of our preferred genAIs got this wrong, even when we varied the prompts to try to help them. Always check genAI results.]

2.4 Displaying Text with `printf`

The **`System.out.printf` method** displays formatted data—the "`f`" in `printf` means "formatted." Figure 2.4 uses `printf` to output `"Welcome to"` and `"Java Programming!"` on separate lines.

Click here to view code image

```
1  // Fig. 2.4: Welcome4.java
2  // Displaying multiple lines with method System.out.printf.
3
4  public class Welcome4 {
5     // main method begins program execution
6     public static void main(String[] args) {
7        System.out.printf("%s%n%s%n", "Welcome to", "Java Programming!");
8     } // end method main
9  } // end class Welcome4
```

```
Welcome to
Java Programming!
```

Fig. 2.4 | Displaying multiple lines with method `System.out.printf`.

Line 7 calls `System.out.printf` to display the program's output. The method call specifies three arguments. When a method requires two or more arguments, you place them in a **comma-separated list** with a space after each comma (`,`) for readability.

Method `printf`'s first argument is a **format string** that may consist of **fixed text** and **format specifiers**. Fixed text is output by `printf` just as it would be by `print` or `println`. Each format specifier is a placeholder for a value and specifies the type of data to output.

Format specifiers begin with a percent sign (`%`) followed by a character representing the data type. The **`%s` format specifier** is a placeholder for a string. The format string in line 7 specifies that `printf` should output two strings, each followed by a newline character. This example substitutes `"Welcome to"` for the first `%s` and `"Java Programming!"` for the second `%s`. The output shows that two lines of text are displayed.

Notice that instead of the newline escape sequence `\n`, which we've been using with `System.out.print` and `System.out.println`, we used the **`%n` format specifier**, which may appear only in format strings. In later chapters, we'll present additional details of

formatting output with `printf`, and Appendix B covers print formatting in more depth.

Generative AI

1 Prompt genAIs to write one Java `printf` statement that produces the following output:

```
#
##
###
####
#####
```

Compare the generated code to the following statement:

Click here to view code image

```
System.out.printf("#%n##%n###%n####%n#####%n");
```

2.5 Another Program: Adding Integers

The program in Fig. 2.5 reads (or inputs) two **integers** (whole numbers, such as –22, 7, 0 and 1024) entered by a user, stores them in variables, computes their sum and displays it. In our program outputs, we use bold text to identify user inputs (**`45`** and **`72`** in this example).

Click here to view code image

```
1  // Fig. 2.5: Addition.java
2  // Addition program that inputs two numbers then displays their sum.
3  import java.util.Scanner; // program uses class Scanner
4
5  public class Addition {
6     // main method begins program execution
7     public static void main(String[] args) {
8        // create a Scanner to obtain input from the user
9        Scanner input = new Scanner(System.in);
10
11       System.out.print("Enter first integer: "); // prompt
12       int number1 = input.nextInt(); // read first number from user
13
```

```
14          System.out.print("Enter second integer: "); // prompt
15          int number2 = input.nextInt(); // read second number from user
16
17          int sum = number1 + number2; // add numbers, then store total in sum
18
19          System.out.printf("Sum is %d%n", sum); // display sum
20       } // end method main
21    } // end class Addition
```

```
Enter first integer: 45
Enter second integer: 72
Sum is 117
```

Fig. 2.5 | Addition program that inputs two numbers then displays their sum.

`import` Declarations

A great strength of Java is its rich set of predefined classes that you can reuse rather than "reinventing the wheel." These classes are grouped into **packages**—named groups of related classes—collectively called the Java Application Programming Interface (Java API).

Line 3 is an **`import` declaration** that helps the compiler locate a class used in this program. It indicates that the program uses the predefined `Scanner` class (discussed shortly) from the **`java.util` package**. The compiler then ensures that you use the class correctly.

All `import` declarations must appear before you declare a class in the file. Placing an `import` declaration inside or after a class declaration is a syntax error. Forgetting to include an `import` declaration for a class that must be `import`ed results in a compilation error containing a message such as "`cannot find symbol`." When this occurs, check that you provided the proper `import` declarations and that the names in them are correct, including proper capitalization.

Declaring and Creating a `Scanner` to Obtain User Input

You must declare each variable with a name and a type before using it. A **variable's name** enables the program to access the variable's value in memory. By convention, variable-name identifiers use the camel-case naming convention with a lowercase

first letter—for example, `firstNumber`. A variable name can be any valid identifier—again, a series of characters consisting of letters, digits, underscores (_) and dollar signs ($) that does not begin with a digit and does not contain spaces. A **variable's type** specifies what kind of information is stored at that location in memory. Like other statements, declaration statements end with a semicolon (;).

Line 9 of `main` is a **variable declaration statement** that specifies the name (`input`) and type (`Scanner`) of a variable used in this program. A **`Scanner`** (package `java.util`) enables a program to read data (such as, numbers and strings) for use in a program. The data can come from many sources, such as the user or a file on disk. Before using a `Scanner`, you must create it and specify the source of the data.

The `=` in line 9's declaration indicates that `Scanner` variable `input` should be initialized to prepare it for use in the program. The initial value is the result of the expression to the right of the equals sign—`new Scanner(System.in)`. This expression uses the **`new` keyword** to create a `Scanner` object that reads characters entered by the user. The **`System.in` standard input object** enables programs to read bytes entered by the user. The `Scanner` translates these bytes into types (like `int`) the program can use.

Prompting the User for Input

Line 11 uses `System.out.print` to display `"Enter first integer: "`. This **prompt message** directs the user to take a specific action. We use `print` here rather than `println` so that the user's input appears on the same line as the prompt—the space after the prompt's colon (:) makes the program's output more readable. Recall from Section 2.2 that identifiers starting with capital letters typically represent class names. The `System` class is part of the **`java.lang` package**, which Java automatically imports into every program. The `java.lang` classes are the only Java API classes that do not require an `import` declaration.

Declaring a Variable to Store an Integer and Obtaining One from the User

The variable declaration statement in line 12 declares that `number1` holds data of **type `int`**—that is, integer (whole number) values, such as `72`, `-1127` and `0`. The value range for an `int` is –2,147,483,648 to +2,147,483,647. A program's `int` values may not contain commas because Java uses them to separate items in lists,

such as the arguments to method `System.out.printf` in line 19 of this program. For readability, you can group digits with underscores as a **digit separator**—so `60_000_000` represents the value 60,000,000.

Types `float` and `double` hold real numbers, and **type `char`** holds character data. Real numbers contain decimal points, such as `3.4`, `0.0` and `-11.19`. Variables of type `char` represent individual characters, such as

- a letter (e.g., `A` or `z`),
- a digit (`0`–`9`),
- a special character (e.g., `*` or `%`) or
- an escape sequence (e.g., `\t` for a tab or `\n` for a newline).

The types `int`, `float`, `double` and `char` are **primitive types**. Their names are keywords and must appear in all lowercase letters. The other primitive types are `boolean`, `byte`, `short` and `long`. For a summary of the numeric primitive types' value ranges, visit the Java Language Specification Section 4.2, "Primitive Types and Values."[6]

6. The Java Language Specification, Java SE 22 Edition. Section 4.2, "Primitive Types and Values" Accessed May 30, 2024. `https://docs.oracle.com/javase/specs/jls/se22/html/jls-4.html#jls-4.2`.

The `=` in line 12 indicates that `int` variable `number1` should be initialized in its declaration with the result of `input.nextInt()`. This uses the `Scanner` object `input`'s `nextInt` method to obtain an integer from the user. At this point, the program waits for the user to enter a number and press the *Enter* key to submit it to the program.

Our program assumes that the user enters a valid integer value. If not, a logic error will occur, and the program will terminate. Chapter 10, Exception Handling: A Deeper Look, discusses how to make your programs more robust by enabling them to handle such errors. This helps make your program fault-tolerant.

Obtaining a Second Integer

Line 14 prompts the user to enter the second integer. Line 15 declares the `int` variable `number2` and initializes it with a second integer read from the user.

Using Variables in a Calculation

Line 17 declares the `int` variable `sum` and initializes it with the result of the calculation `number1 + number2`. The `+` operator is a **binary operator** because it has two **operands**—`number1` and `number2`. Portions of statements that contain calculations are called **expressions**. In fact, an expression is any portion of a statement that has a value. The value of the expression `number1 + number2` is their sum. Similarly, the value of the expression `input.nextInt()` (lines 12 and 15) is the integer entered by the user. Place spaces on either side of a binary operator for readability.

Displaying the Calculation Result

After the calculation has been performed, line 19 uses method `System.out.printf` to display `sum`'s value. The **`%d` format specifier** is a placeholder for an `int` value (`sum`)—the letter `d` stands for "decimal (base 10) integer." The remaining characters in the format string are all fixed text. So, method `printf` displays `"Sum is "`, followed by `sum`'s value—which replaces `%d`—and a newline.

We could have combined lines 17 and 19 into the single statement

Click here to view code image

```
System.out.printf("Sum is %d%n", (number1 + number2));
```

The parentheses around the expression `number1 + number2` are redundant—they're included to emphasize that the value of the entire expression is output in the position of the `%d` format specifier.

Java API Documentation

For each new Java API class we present, we show the name of its package. This information helps you locate descriptions of each package and class in the Java API documentation. The documentation can be found at

Click here to view code image

```
https://docs.oracle.com/en/java/javase/23/docs/api/index.html
```

The following page overviews how to use the API documentation

Click here to view code image

```
https://docs.oracle.com/en/java/javase/23/docs/api/help-
   doc.html#overview
```

You may download the documentation for offline use from

Click here to view code image

```
https://www.oracle.com/java/technologies/javase-jdk23-doc-
   downloads.html
```

Declaring and Initializing Variables in Separate Statements

Each variable must have a value before you use it in a calculation (or other expression). The variable declaration statement in line 12 declares `number1` and initializes it with a value the user enters. Sometimes, you declare a variable in one statement then give it a value in another. For example, line 12 could have been written in two statements as

Click here to view code image

```
int number1; // declare the int variable number1
number1 = input.nextInt(); // assign the user's input to number1
```

The first statement declares `number1` but does not initialize it. The second statement uses the **assignment operator** (`=`) to assign `number1` the integer entered by the user. You can read this statement as "`number1` gets the value of `input.nextInt()`." Everything to the right of the `=` is evaluated before the assignment stores the result.

Generative AI

1 Prompt genAIs to write a Java program that reads three integers from the user, calculates their sum and displays the result. Compare the code they write to the addition program in this section. Execute the code to confirm that it works. If you get an error, prompt them with the error message and ask them to fix the code. Then, run it again.
2 Prompt genAIs to create a brief tutorial that tells you how to use the Java API documentation.

2.6 Arithmetic

Many programs perform arithmetic calculations using the **arithmetic operators** summarized in the following table:

Operation	Operator	Algebraic expression	Arithmetic expression
Addition	`+`	$f + 7$	`f + 7`
Subtraction	`-`	$p - c$	`p - c`
Multiplication	`*`	bm	`b * m`
Division	`/`	x / y or $x \div y$	`x / y`
Remainder	`%`	$r \bmod s$	`r % s`

Note the use of various special symbols not used in algebra. The **asterisk** (`*`) indicates multiplication, and the **percent sign** (`%`) is the **remainder operator**, which we'll discuss shortly. The arithmetic operators in the table are binary operators—each operates on two operands—e.g., the expression `f + 7` contains the binary operator `+` and the two operands `f` and `7`.

Integer Division

Integer division yields an integer quotient. For example, the expression `7 / 4` evaluates to `1`, and the expression `17 / 5` evaluates to `3`. Any fractional part in integer division is truncated (that is, discarded)—no rounding occurs.

Remainder Operator (`%`)

The remainder operator, `%`, yields the remainder after division—`x % y` yields the remainder after dividing `x` by `y`. Thus, `7 % 4` yields `3`, and `17 % 5` yields `2`. This operator is most commonly used with integer operands but also can be used with other arithmetic types. Popular uses of this operator are to determine whether an integer is even or odd, or whether one number is a multiple of another.

Arithmetic Expressions in Straight-Line Form

Arithmetic expressions must be written in **straight-line form** to facilitate entering programs into computers. Thus, expressions such as "`a` divided by `b`" must be written as `a/b` so all constants, variables and operators appear in a straight line. The following algebraic notation is generally not acceptable to compilers:

ab

Parentheses for Grouping Subexpressions

Parentheses group terms in Java expressions in the same manner as in algebraic expressions. For example, to multiply `a` times the quantity `b + c`, we write

```
a * (b + c)
```

If an expression contains **nested parentheses**, such as

```
(a * (b + c))
```

the expression in the innermost set of parentheses—`b + c` in this case—evaluates first.

Rules of Operator Precedence

Java applies the arithmetic operators in a precise sequence determined by the **rules of operator precedence**, which are generally the same as those followed in algebra:

- Multiplication, division and remainder operations are applied first. If an expression contains several such operations, they're applied from left to right. Multiplication, division and remainder operators have the same level of precedence.
- Addition and subtraction operations are applied next. If an expression contains several such operations, the operators are applied from left to right. Addition and subtraction operators have the same level of precedence.
- The assignment operator is applied last.

These rules enable Java to apply operators in the correct order.[7] When we say operators are applied from left to right, we're referring to their **grouping**. Some group from right to left. The following table summarizes these rules of operator precedence:

7. We use simple examples to explain the order of evaluation. Subtle order-of-evaluation issues can occur in more complex expressions. For more information, refer to Chapter 15 of The Java™ Language Specification (`https://docs.oracle.com/javase/specs/jls/se22/html/jls-15.html`).

Operator(s)	Evaluation (precedence)
* / %	Multiplication Division Remainder	Multiplied first. If there are several operators of this type, they're evaluated from left to right.

Remainder	Evaluated next. If there are several operators of this type, they're evaluated from left to right.
Addition	
Subtraction	
Assignment	Assigned last.

You can view the complete operator precedence chart for Java's operators at

Click here to view code image

```
https://docs.oracle.com/javase/tutorial/java/nutsandbolts/
   operators.html
```

Generative AI

1 Prompt genAIs to explain the order in which Java evaluates the operators in the following statement and to specify `x`'s value after the statement executes:

Click here to view code image

```
int x = 2 % 2 + 2 * 2 - 2 / 2;
```

The genAIs should say that the order of evaluation is

```
% * / + -
```

and the value assigned to `x` is `3`.

2.7 Decision Making: Equality and Relational Operators

A condition is an expression that can be **`true`** or **`false`**. The **`if` statement** makes a decision based on a condition's value. You form conditions in `if` statements using the **equality operators** (**`==`** and **`!=`**) and **relational operators** (**`>`**, **`<`**, **`>=`** and **`<=`**) summarized below:

Algebraic equality or relational operator	Java equality or relational operator	Sample Java condition	Meaning of Java condition
Equality operators			
=	==	x == y	x is equal to y

≠ is not equal to `y`

Relational operators

> is greater than `y`

< is less than `y`

≥ is greater than or equal to `y`

≤ is less than or equal to `y`

Both equality operators have the same precedence level, which is lower than that of the relational operators. The equality operators group from left to right. The relational operators all have the same precedence level and also group from left to right.

Using the `if` Statement

Figure 2.6 uses six `if` statements to compare two integers input by the user. If the condition in any of these is `true`, the corresponding body statement executes; otherwise, it is skipped. We use a `Scanner` to input the integers from the user and store them in variables `number1` and `number2`. The program compares the numbers with all six relational and equality operators and displays the results of the comparisons that are `true`. We show three sample outputs for different pairs of values entered by the user.

Click here to view code image

```
1  // Fig. 2.6: Comparison.java
2  // Compare integers using if statements, relational operators
3  // and equality operators.
4  import java.util.Scanner; // program uses class Scanner
5
6  public class Comparison {
7     // main method begins program execution
8     public static void main(String[] args) {
9        // create Scanner to obtain input from the user
10       Scanner input = new Scanner(System.in);
11
12       System.out.print("Enter first integer: "); // prompt
13       int number1 = input.nextInt(); // read first number from user
14
15       System.out.print("Enter second integer: "); // prompt
16       int number2 = input.nextInt(); // read second number from user
17
18       if (number1 == number2) {
19          System.out.printf("%d == %d%n", number1, number2);
20       }
21
22       if (number1 != number2) {
```

```
23             System.out.printf("%d != %d%n", number1, number2);
24          }
25
26          if (number1 < number2) {
27             System.out.printf("%d < %d%n", number1, number2);
28          }
29
30          if (number1 > number2) {
31             System.out.printf("%d > %d%n", number1, number2);
32          }
33
34          if (number1 <= number2) {
35             System.out.printf("%d <= %d%n", number1, number2);
36          }
37
38          if (number1 >= number2) {
39             System.out.printf("%d >= %d%n", number1, number2);
40          }
41       } // end method main
42    } // end class Comparison
```

```
Enter first integer: 777
Enter second integer: 777
777 == 777
777 <= 777
777 >= 777
```

```
Enter first integer: 1000
Enter second integer: 2000
1000 != 2000
1000 < 2000
1000 <= 2000
```

```
Enter first integer: 2000
Enter second integer: 1000
2000 != 1000
2000 > 1000
2000 >= 1000
```

Fig. 2.6 | Compare integers using `if` statements, relational operators and equality operators.

Line 10 in class `Comparison`'s `main` method sets up a `Scanner` for reading data from the user. Lines 12–13 prompt the user to enter the first integer, then input the value and store it in the `int` variable `number1`. Lines 15–16 prompt the user to enter the second

integer, then input the value and store it in the `int` variable `number2`. Lines 18–20 compare the `number1` and `number2` values to determine whether they're equal. If so, line 19 displays the numbers separated by `==`. The `if` statements starting in lines 22, 26, 30, 34 and 38 compare `number1` and `number2` using the operators `!=`, `<`, `>`, `<=` and `>=`. For those `if` statements in which the conditions are `true`, the corresponding body statement displays an appropriate line of text showing the condition that evaluated to `true`.

Blocks

Each `if` statement in Fig. 2.6 contains one body statement that's indented to enhance readability. IDEs typically do this indentation for you. Also, notice that we've enclosed each body statement in a pair of braces, `{ }`, creating a **block**.

You don't need to use braces around single-statement bodies, but you must include the braces around multiple-statement bodies. You'll see later that forgetting to enclose multiple-statement bodies in braces leads to errors. To avoid such errors, always enclose an `if` statement's body statement(s) in braces.

Placing a semicolon immediately after the right parenthesis after the condition in an `if` statement is often a logic error (although not a syntax error). The semicolon (`;`) acts as an **empty statement** (i.e., a statement that does nothing). In this case, the `if` statement performs no action, regardless of whether the condition is `true`. Worse yet, the indented statement that appears to be in the `if` statement's body always executes—a logic error that typically causes the program to produce incorrect results.

Whitespace

Note the use of whitespace in Fig. 2.6. The compiler ignores most whitespace, so statements may be split over several lines and spaced according to your preferences without affecting a program's meaning. Identifiers and strings may not be split. Ideally, statements should be kept small, but this is not always possible.

If you spread a statement over several lines, choose natural breaking points, such as after a comma in a comma-separated list or after an operator in a lengthy expression. When a statement is split across two or more lines, indent all lines after the first by the same amount until the end of the statement.

Generative AI

1 Prompt genAIs to find and correct the error(s) in the following `if` statements:

Click here to view code image

```
if (c < 7); {
   System.out.println("c is less than 7");
}
if (c => 7) {
   System.out.println("c is greater than or equal to 7");
}
```

Compare their results to the following: In the first `if` statement, there's a semicolon after the `if` conditions right parenthesis, so the output statement executes regardless of whether the condition is `true`. To correct this, the genAIs should remove the semicolon after the right parenthesis. In the second `if` statement, the symbol `=>` in the condition is not a valid relational operator. The genAIs should replace `=>` with `>=`.

2.8 Objects-Natural Case Study: Creating and Using Objects of the Java API's `String` Class

Throughout this book, we emphasize using valuable preexisting classes from the Java API and various libraries from the Java open-source community. You'll focus on knowing what libraries are out there, choosing the ones you'll need for your applications, creating objects from existing library classes and making those objects exercise their capabilities. By "objects natural," we mean that you'll be able to conveniently create and program with powerful objects even before you learn to create your own custom classes in Chapter 8.

You've already worked with Java objects. You used the `System.out` object to display a program's output, and you used `Scanner` objects with the `System.in` object to obtain inputs from the user. These encapsulate the mechanisms for output and input. The `System.out` and `System.in` objects are created for you by the

`System` class, and you created the `Scanner` objects with the `new` operator. In this section, you'll create and interact with objects of the Java API's **`String` class**.[8]

8. You'll learn additional `String` capabilities in subsequent chapters. Chapter 7 discusses class `String` in detail, test-driving many more of its methods.

Test-Driving Class `String`

Classes cannot execute by themselves. A `Person` object can drive a `Car` object by telling it what to do (go faster, go slower, turn left, turn right, etc.) without knowing how the car's internal mechanisms work. Similarly, the `main` method can "drive" `String` objects by calling their methods—without knowing how the class is implemented. In this sense, `main` in the following program is referred to as a **driver program**. Figure 2.7's `main` method test-drives several `String` objects.

Click here to view code image

```
1  // Fig. 2.7: StringTest.java
2  // String class test program.
3  public class StringTest {
4     public static void main(String[] args) {
5        String s1 = "happy";
6        String s2 = " birthday";
7        String s3 = ""; // creates an empty string
8
9        // display the Strings and show their lengths
10       System.out.printf("s1: \"%s\"; length: %d%n", s1, s1.length());
11       System.out.printf("s2: \"%s\"; length: %d%n", s2, s2.length());
12       System.out.printf("s3: \"%s\"; length: %d%n%n", s3, s3.length());
13
14       // compare Strings
15       System.out.println("Results of comparing Strings:");
16       System.out.printf("s1.equals(\"happy\"): %b%n", s1.equals("happy"));
17       System.out.printf("s2.equals(s1): %b%n%n", s2.equals(s1));
18
19       // test String method isEmpty
20       System.out.println("Testing s3.isEmpty():");
21
22       if (s3.isEmpty()) {
23          System.out.println("s3 is empty; assigning to s3");
24          s3 = s1 + s2; // assign s3 the result of concatenating s1 and s2
25          System.out.printf("s3: \"%s\"; length: %d%n%n", s3, s3.length());
26       }
27
28       // testing whether Strings start with "ha" or end with "ay"
```

```
29          System.out.printf("s1.startsWith(\"ha\"): %b%n",
30             s1.startsWith("ha"));
31          System.out.printf("s2.startsWith(\"ha\"): %b%n",
32             s2.startsWith("ha"));
33          System.out.printf("s1.endsWith(\"ay\"): %b%n", s1.endsWith("ay"));
34          System.out.printf("s2.endsWith(\"ay\"): %b%n", s2.endsWith("ay"));
35       } // end method main
36    } // end class StringTest
```

```
s1: "happy"; length: 5
s2: " birthday"; length: 9
s3: ""; length: 0

Results of comparing Strings:
s1.equals("happy"): true
s2.equals(s1): false

Testing s3.isEmpty():
s3 is empty; assigning to s3
s3: "happy birthday"; length: 14

s1.startsWith("ha"): true
s2.startsWith("ha"): false
s1.endsWith("ay"): false
s2.endsWith("ay"): true
```

Fig. 2.7 | `String` class test program.

Creating `String` Objects

For each string literal, the Java compiler creates an object of the Java API's `String` class and initializes it with the literal's characters.[9] So, lines 5–7 create three `String` objects: `s1` contains `"happy"`, `s2` contains `" birthday"`, and `s3` contains the **empty string (`""`)**. When we declare `int` variables, as we did earlier, the compiler knows what `int` is—it's a primitive type built into Java. Similarly, the compiler understands aspects of the Java API's `String` class, such as how to create a `String` object from a string literal.

9. For most classes, you must create objects of the class using `new`, as you did with class `Scanner` in Figs. 2.5 and 2.6.

Like the `System` class, the `String` class is part of the `java.lang` package, so it's imported into every program. When packaged properly, classes can be reused by other programmers. This is one of the most significant benefits of working with languages like Java that support object-oriented programming. Such languages have

rich libraries of powerful prebuilt classes. For example, you can reuse the Java API's classes in any program simply by importing them, as we did with the `Scanner` class in Figs. 2.5 and 2.6.

`String` Method `length`

Lines 10–12 output each `String` and its length. The `String` class's **`length` method** returns the number of characters stored in a particular `String` object. To call this method for a specific object, you specify the object's name (`s1`), followed by the **dot operator** (`.`), then the method name (`length`) and a set of parentheses. Empty parentheses indicate that `length` does not require additional arguments to perform its task.

Comparing `String` Objects

`String`s can be compared with one another for equality using the `String` class's **`equals` method**, which returns `true` or `false`:[10]

- Line 16 compares `s1`'s contents to the contents of the string literal `"happy"`—these `String`s contain the same characters, so `equals` returns `true`.
- Line 17 compares `s2`'s contents to `s1`'s contents—these `String`s do not contain the same characters so `equals` returns `false`.

10. Note that operator `==` cannot be used to compare the contents of `String` objects. We'll explain why in the object-oriented programming chapters.

`String` comparisons are case-sensitive.[11] The `printf` calls in lines 16 and 17 use the **`%b` format specifier**, which is a placeholder for the value of a condition. This format specifier causes `printf` to display a condition's value as the word `"true"` or the word `"false"`.

11. In Chapter 7, you'll see that `String`s perform "lexicographical" comparisons using the numerical values of the characters in each `String`.

`String` Method `isEmpty`

Line 22 calls `String`'s **`isEmpty` method**, which returns `true` if the `String` is empty—i.e., its length is `0`. The object `s3` was initialized to the empty string (`""`), so it is indeed empty, and the body of the `if` statement will execute.

`String` Concatenation

Line 24 assigns a new value to `s3` produced by "adding" the `String`s `s1` and `s2` using the `+` operator—known as **`String` concatenation**. After the assignment, `s3` contains the characters of `s1` followed by the characters of `s2`—that is, `"happy birthday"`. Line 25 outputs `s3`'s contents and length to demonstrate that the `String` concatenation worked correctly.

`String` Methods `startsWith` and `endsWith`

Lines 29–34 demonstrate **`String`'s `startsWith` method** and **`endsWith` method**, which return `true` if the `String` starts with or ends with a specified substring, respectively; otherwise, they return `false`. These `String` comparisons are also case-sensitive. Lines 30 and 32 show that `s1` starts with the substring `"ha"`, but `s2` does not. Lines 33 and 34 show that `s1` does not end with `"ay"` but `s2` does.

Generative AI

1 Prompt genAIs to write a Java program that inputs a first name and a last name individually then displays `"Welcome to Java,"` followed by the first name, a space, the last name and an exclamation point (`"!"`). Run the generated code to ensure it works. What new `Scanner` method do the genAIs introduce to read a `String` from the user?

2.9 Wrap-Up

In this chapter, you created basic Java programs. We started with programs that displayed messages on the screen using `System.out`'s `print`, `println` and `printf` methods. The addition program used a `Scanner` object to perform input from the user. We discussed the arithmetic operators (`+`, `-`, `*`, `/`, and `%`) and how the operator precedence rules determine the order in which Java evaluates operators in an expression. We briefly mentioned other primitive types. You used the `if` statement to make decisions based on conditions formed with the equality (`==` and `!=`) and relational (`<`, `<=`, `>` and `>=`) operators.

Finally, we introduced our objects-natural approach to learning Java by creating objects of the powerful Java API class `String` and

interacting with them. Throughout the chapter, you used generative AI to leverage your Java learning experience. In subsequent chapters, you'll create and use many objects of existing classes to accomplish significant tasks with minimal amounts of code. Then, in Chapters 8–9, you'll use Java features that enable you to craft valuable custom classes. In the next chapter, we introduce control statements that specify the order in which a program performs its actions.

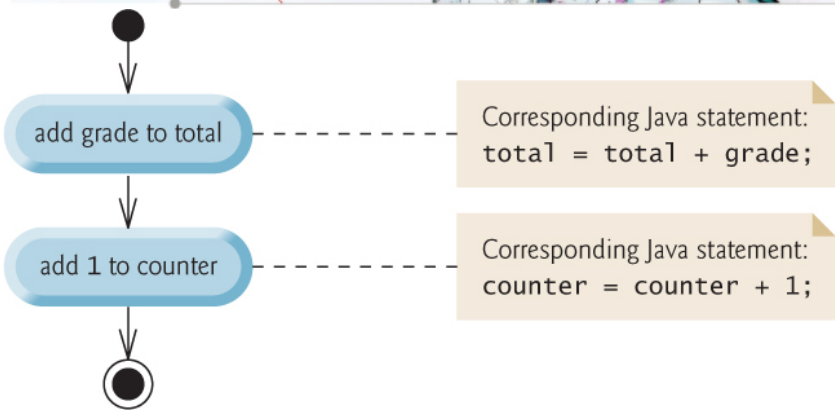
Special Section: Generative AI Exercises

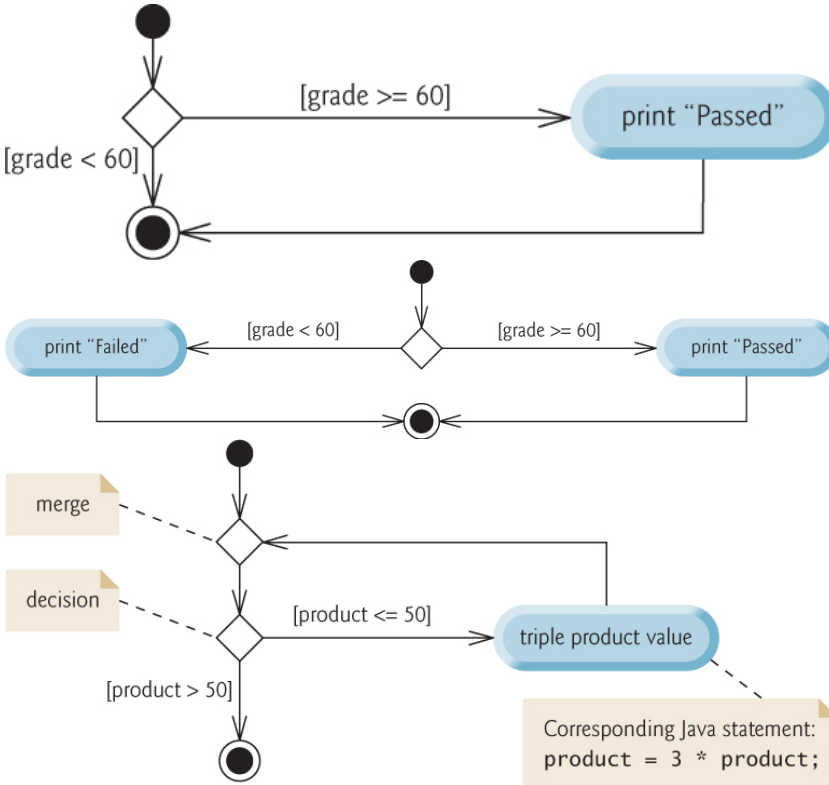
For each of the following questions, try multiple genAIs to determine which performs best. Always verify the results. Keep in mind that they sometimes make mistakes and even hallucinate, creating unreliable information.

2.1 ***(Commenting Java Code)*** Prompt genAIs to add comments to uncommented code then check whether the comments are relevant.

2.2 ***(Improving Java Code Comments)*** Prompt genAIs to improve comments in a Java program.

2.3 ***(Comparing Your Code with Generative AI Code)*** For each program you write, prompt genAIs to create the same program. Compare their generated code to yours. Tell the genAIs what Java features have been presented so far and ask them to limit their code to those features.


add grade to total
Corresponding Java statement:
total = total + grade;
add 1 to counter
Corresponding Java statement:
counter = counter + 1;



Control Statements: Part 1

Objectives

In this chapter, you'll:

- Use the `if` and `if...else` selection statements to choose between alternative actions.
- Use the `while` iteration statement to execute statements in a program repeatedly.
- Use counter-controlled iteration, sentinel-controlled iteration and nested control statements.
- Use the compound assignment operators and the increment and decrement operators.
- Consider the portability of primitive types.
- Use generative AI to leverage your Java learning experience.
- Continue our objects-natural approach with a case study on creating and manipulating integers as large as you want them to be.

Outline

3.1 Introduction

3.2 Control Structures

3.2.1 Sequence Structure

3.2.2 Selection Statements

3.2.3 Iteration Statements

3.2.4 Summary of Control Statements

3.3 `if` Single-Selection Statement

3.4 `if...else` Double-Selection Statement

3.4.1 Nested `if...else` Statements

3.4.2 Blocks

3.4.3 Conditional Operator (`?:`)

3.5 `while` Iteration Statement

3.6 Counter-Controlled Iteration

3.6.1 Implementing the Algorithm

3.6.2 Integer Division and Truncation

3.6.3 Arithmetic Overflow

3.6.4 Input Validation

3.7 Sentinel-Controlled Iteration

3.7.1 Implementing Sentinel-Controlled Iteration

3.7.2 Explicitly and Implicitly Converting Between Primitive Types
3.7.3 Formatting Floating-Point Numbers
3.8 Nested Control Statements
3.8.1 Problem Statement
3.8.2 Implementing the Program
3.9 Compound Assignment Operators
3.10 Increment and Decrement Operators
3.11 Primitive Types
3.12 Objects-Natural Case Study: Super-Sized Integers
3.13 Wrap-Up
Special Section: Generative AI Exercises

3.1 Introduction

In this chapter and the next, we present the theory and principles of structured programming. As you'll see in later chapters, the concepts presented here will be crucial in building classes and manipulating objects.

We discuss the `if` statement in more detail and introduce the `if`...`else` and `while` statements. We introduce the compound assignment operators and the increment and decrement operators. We also consider the portability of Java's primitive types. Finally, we continue our objects-natural approach with a case study on creating and manipulating super-sized integers that can represent values beyond the limited integer ranges supported by Java's integer primitive types.

3.2 Control Structures

Statements in a program typically execute one after the other in the order they're written, as you saw in Chapter 2. This process is called **sequential execution**. Various Java statements enable you to specify that the next statement to execute is not necessarily the next one in sequence. This is called **transfer of control**.

During the 1960s, it became clear that the indiscriminate use of transfers of control was the root of much difficulty experienced by

software development groups. The blame was pointed at the **`goto` statement** (used in most programming languages of the time), which allowed you to specify a transfer of control to one of a wide range of destinations in a program.[1]

1. Java does not have a `goto` statement; however, `goto` is reserved by Java and should not be used as an identifier in programs. Using it causes a compilation error.

The research of Bohm and Jacopini[2] demonstrated that programs could be written without any `goto` statements. The challenge for the era's programmers was to shift their styles to "`goto`-less programming." The term **structured programming** became almost synonymous with "`goto` elimination." Not until the 1970s did most programmers start taking structured programming seriously. The results were impressive. Software development groups reported shorter development times, more frequent on-time delivery of systems and more frequent within-budget completion of software projects. The key to these successes was that structured programs were clearer, easier to debug and modify, and more likely to be bug-free in the first place.

2. C. Bohm and G. Jacopini, "Flow Diagrams, Turing Machines, and Languages with Only Two Formation Rules," *Communications of the ACM*, Vol. 9, No. 5, May 1966, pp. 366–371.

Bohm and Jacopini's work demonstrated that all programs could be written in terms of only three control structures—the sequence structure, the selection structure and the iteration structure. When we introduce Java's control-structure implementations, we'll refer to them in the terminology of the *Java Language Specification* as "control statements."

3.2.1 Sequence Structure

The **sequence structure** is built into Java. Unless directed otherwise, the computer executes Java statements one after the other in the order they're written—that is, in sequence. The following **UML activity diagram** illustrates a typical sequence structure in which two calculations are performed in order:
Java lets you have as many actions as you want in sequence. As we'll soon see, anywhere a single action may be placed, we may place several actions in sequence.

A UML activity diagram models the **workflow** (also called the **activity**) of a portion of a software system. Such workflows may include a portion of an algorithm, like the sequence structure

above. Activity diagrams are composed of symbols, such as **action-state symbols** (rectangles with their left and right sides replaced with outward arcs), **diamonds** and **small circles**. These symbols are connected by **transition arrows**, which represent the flow of the activity—that is, the order in which the actions should occur.

Activity diagrams help you develop and represent algorithms. They clearly show how control structures operate. In Chapters 3 and 4, we use the UML to show each control statement's workflow —also called its **flow of control**.

The preceding activity diagram includes two **action states**, each containing an **action expression** specifying a particular action to perform—"add grade to total" or "add 1 to counter." Other actions might include calculations or input/output operations. The arrows represent transitions, indicating the order in which the actions described by the action states occur. The program that implements the preceding diagram's activities first adds `grade` to `total`, then adds `1` to `counter`.

The **solid circle** at the top of the activity diagram represents the **initial state**—the beginning of the workflow before the program performs the modeled actions. The **solid circle surrounded by a hollow circle** at the bottom of the diagram represents the **final state**—the end of the workflow after the program performs the modeled actions.

The diagram also includes rectangles with the upper-right corners folded over. These optional **UML notes** (like comments in Java) are explanatory remarks describing the purpose of the diagram's symbols. Our notes show the Java code associated with each action state. A **dotted line** connects each note with the element it describes. Activity diagrams usually do not show the corresponding Java code. We show it here to illustrate how the diagram relates to Java code. For more information on the UML, visit `https://uml.org`.

3.2.2 Selection Statements

Java has three types of **selection statements** (discussed in this chapter and Chapter 4). The `if` statement (Section 3.3) either performs (selects) an action if a condition is `true` or skips it if the condition is `false`. The `if...else` statement (Section 3.4) performs an action if a condition is `true` and performs a different action if the condition is `false`. The `switch` statement (Section 4.8) performs one of several actions depending on the value of an

expression.

The `if` statement is called a **single-selection statement** because it selects or ignores a single action (or, as we'll soon see, a single group of actions). The `if`…`else` statement is called a **double-selection statement** because it selects between two different actions (or groups of actions). The `switch` statement is called a **multiple-selection statement** because it selects among many different actions (or groups of actions).

3.2.3 Iteration Statements

Java provides four **iteration statements**—also called **repetition** or **looping statements**—for performing statements repeatedly while a **loop-continuation condition** is `true`. The iteration statements are `while` (Section 3.5), `for` (Section 4.3), `do`…`while` (Section 4.7), and enhanced `for` (Section 6.13). The `while` and `for` statements perform the action (or group of actions) in their bodies zero or more times—if the loop-continuation condition is initially `false`, the action (or group of actions) will not execute. The `do`…`while` statement performs the action (or group of actions) in its body one or more times.

Keywords

The words `if`, `else`, `switch`, `while`, `do` and `for` are Java keywords. We showed the complete list of Java keywords in Section 2.2. You can also find them in the *Java Language Specification*, Section 3.9, "Keywords."[3]

3. The Java Language Specification, Java SE 22 Edition. Section 3.9, "Keywords." Accessed May 21, 2024. https://docs.oracle.com/javase/specs/jls/se22/html/jls-3.html#jls-3.9.

3.2.4 Summary of Control Statements

Java has only three kinds of control structures, which we refer to as control statements from this point forward:

- the sequence statement,
- selection statements (three types) and
- iteration statements (four types).

We can model each control statement as an activity diagram

containing initial and final states representing a control statement's entry and exit points. **Single-entry/single-exit control statements** make it easy to build programs by connecting the exit point of one to the entry point of the next—that is, **control-statement stacking**. The only one other way in which control statements may be connected is **control-statement nesting** in which one control statement appears inside another.

3.3 `if` Single-Selection Statement

We briefly introduced `if` single-selection statement in Section 2.7. Programs use selection statements to choose among alternative courses of action. For example, suppose a passing grade on an exam is 60. The following `if` statement determines whether the condition `grade >= 60` is `true`:

Click here to view code image

```
if (grade >= 60) {
   System.out.println("Passed");
}
```

If so, the statement prints `"Passed"`; otherwise, the condition is `false`, and the output statement is ignored and the next statement in sequence (if any) executes. Indenting the second line is optional but recommended for clarity.

`boolean` Primitive Type

In Section 2.7, you created conditions using the relational and equality operators. A Java condition must evaluate to **`true`** or **`false`**—each is a Java keyword. Java provides the **`boolean` primitive type** for variables that can store only `true` or `false` values.

UML Activity Diagram for an `if` Statement

The following diagram illustrates the single-selection `if` statement's flow of control:
This figure contains an activity diagram's most important symbol—the diamond, or **decision symbol**, which indicates a decision is to

be made. The workflow continues along a path determined by the symbol's associated **guard conditions**, which can be true or false. Each transition arrow emerging from a decision symbol has a guard condition specified in square brackets next to the arrow. If a guard condition is true, the workflow enters the action state to which the transition arrow points. In this diagram, if the grade is greater than or equal to 60, the program prints "Passed," then transitions to the activity's final state. If the grade is less than 60, the program transitions to the final state without printing a message. The `if` statement is a single-entry/single-exit control statement.

3.4 `if…else` Double-Selection Statement

The `if` single-selection statement performs an indicated action only when the condition is `true`. The **`if...else` double-selection statement** allows you to specify an action to perform when the condition is `true` and another action when the condition is `false`. For example, the following `if…else` statement prints `"Passed"` if the student's grade is greater than or equal to 60; otherwise, it prints `"Failed"`:

Click here to view code image

```
if (grade >= 60) {
   System.out.println("Passed");
}
else {
   System.out.println("Failed");
}
```

In either case, after printing occurs, the next statement in sequence (if any) executes. The `if` and `else` bodies are both indented equally. For readability, you should consistently apply your indentation convention throughout your programs. If there are several indentation levels, each should be indented with the same additional amount of space. We prefer three-space indents. IDEs typically auto-indent the code for you.

UML Activity Diagram for an `if…else` Statement

The following diagram illustrates the flow of control in the preceding `if…else` statement:

Once again, the UML activity diagram begins with the initial state, contains transition arrows, a decision and action states, and ends with a final state.

Generative AI

1 Prompt genAIs with the following code, asking them to find and correct the error(s):

Click here to view code image

```
if (count > 10) {
   System.out.println("count is greater than 10");
}
else
   System.out.println("count is less than or equal to 10");
}
```

Confirm whether the genAIs recognize the missing left brace, `{`, following the `else` keyword.

3.4.1 Nested `if...else` Statements

You can test multiple cases by placing `if...else` statements inside `if...else` statements to create **nested `if...else` statements**. The following nested `if...else` prints `A` for grades greater than or equal to 90, `B` for grades 80 to 89, `C` for grades 70 to 79, `D` for grades 60 to 69 and `F` for all other grades. We use shading to highlight the nesting:

Click here to view code image

```
if (grade >= 90) {
   System.out.println("A");
}
else {
   if (grade >= 80) {
      System.out.println("B");
   }
   else {
      if (grade >= 70) {
         System.out.println("C");
      }
      else {
         if (grade >= 60) {
            System.out.println("D");
         }
```

```
            else {
               System.out.println("F");
            }
         }
      }
   }
```

If `grade` is greater than or equal to 90, the first four conditions in the nested `if`...`else` statement are `true`. However, only the first `if` part's statement executes—the `else` part of the "outermost" `if`...`else` statement is skipped.

Most programmers prefer to write the preceding nested `if`...`else` statement as

```
if (grade >= 90) {
   System.out.println("A");
}
else if (grade >= 80) {
   System.out.println("B");
}
else if (grade >= 70) {
   System.out.println("C");
}
else if (grade >= 60) {
   System.out.println("D");
}
else {
   System.out.println("F");
}
```

The latter form avoids deep indentation to the right, which often leaves little room on a line of source code, forcing lines to be split and making the code less readable. In a nested `if`...`else` statement, ensure that you test for all possible cases.

3.4.2 Blocks

The `if` statement expects only one statement in its body. To include several statements in an `if`'s or `else`'s body, you must enclose the statements in braces (`{` and `}`). Recall from Section 2.7 that statements in braces form a block. You can place a block anywhere in a method where you can place a single statement. We always enclose control statement bodies in braces. This avoids a logic error called the "dangling-`else`" problem.

The following example includes a multi-statement block in an

`if...else` statement's `else` part:

Click here to view code image

```
if (grade >= 60) {
   System.out.println("Passed");
}
else {
   System.out.println("Failed");
   System.out.println("You must take this course again");
}
```

If `grade` is less than 60, both statements in the `else`'s body execute, displaying:

Click here to view code image

```
Failed
You must take this course again
```

The braces surrounding the two statements in the `else` clause are important. Without the braces, the statement

Click here to view code image

```
System.out.println("You must take this course again");
```

would be outside the `else`'s body and `"You must take this course again"` would print even for passing `grade`s—a logic error.

Syntax Errors and Logic Errors

The compiler catches syntax errors, such as when one brace in a block is left out of the program. A logic error, such as an incorrect calculation, has its effect at execution time. A **fatal logic error** causes a program to fail and terminate prematurely. A **nonfatal logic error** allows a program to continue executing but causes it to produce incorrect results.

Empty Statement

Just as a block can be placed anywhere a single statement can be placed, it's also possible to have an empty statement. Recall from Section 2.7 that the empty statement is represented by placing a semicolon (`;`) where a statement would normally be.

Placing a semicolon after the condition in an `if` or `if...else` statement leads to a logic error in single-selection `if` statements

and a syntax error in double-selection `if`…`else` statements (when the `if`-part contains an actual body statement).

Generative AI

1 Prompt genAIs with the following code and ask them to identify and correct the error(s):

Click here to view code image

```
if (grade < 60)
   System.out.println("Failed");
else
   System.out.println("Passed");
   System.out.println("Great job!");
```

Compare their code updates and explanations to the following: Only the first indented statement following `else` is part of the `else`'s body because the two indented statements are not enclosed in braces. So, this code will print `"Great Job!"` even if the student has a failing grade. Also, it's good practice to enclose even single-statement bodies, like the `if`-part's body, in braces to prevent errors if you add more statements to its body in the future. The code should be:

Click here to view code image

```
if (grade < 60) {
   System.out.println("Failed");
}
else {
   System.out.println("Passed");
   System.out.println("Great job!");
}
```

3.4.3 Conditional Operator (`?:`)

The **conditional operator** (`?:`) can be used in place of simple `if`…`else` statements. The conditional operator is the only **ternary operator**—it takes three operands. Together, the operands and the `?:` symbol form a **conditional expression**. For example, the following statement prints the value of `println`'s conditional-expression argument:

Click here to view code image

```
System.out.println(grade >= 60 ? "Passed" : "Failed");
```

- The operand to the left of the question mark (`?`)—grade `>=` 60—is a condition that evaluates to `true` or `false`.
- The operand between the question mark (`?`) and the colon (`:`)—`"Passed"`—is the conditional expression's value if the condition is `true`.
- The operand after the colon (`:`)—`"Failed"`—is the conditional expression's value if the condition is `false`.

The conditional expression in this statement evaluates to `"Passed"` if `grade >= 60` is `true` or `"Failed"` if it's `false`. Thus, this statement with the conditional operator is essentially the same as the `if`...`else` statement shown at the beginning of Section 3.4. The conditional operator's precedence is low, so it's usually placed in parentheses.

Generative AI

1 Prompt genAIs to write a single `System.out.printf` statement that determines whether the `int` variable `number`'s value is odd or even and prints the result. Tell the genAIs to assume `number` exists and contains 7. Compare their generated code to the following statement and confirm their code produces the same result:

Click here to view code image

```
System.out.printf("%d is %s%n",
   number, number % 2 == 0 ? "even" : "odd");
```

```
7 is odd
```

3.5 `while` Iteration Statement

An iteration statement specifies that a program should repeat an action while some condition remains `true`. Consider a program segment that uses the **`while` iteration statement** to find the first power of 3 larger than 50. After this code executes, `product`

contains the result:

```
int product = 3;
while (product <= 50) {
   product = 3 * product;
}
```

First, we initialize `product` to `3`. Then, the `while` statement executes as follows:

- The program tests the condition `product <= 50`, which is `true` because `product` is `3`. The body statement multiplies `product` by `3` and assigns the result (`9`) to `product`. The loop's first iteration is now complete.
- The program tests the condition again—it's `true` because `product` is now `9`. The body sets `product` to `27`. The loop's second iteration is now complete.
- The program tests the condition again—it's `true` because `product` is now `27`. The body sets `product` to `81`. The loop's third iteration is now complete.
- The program tests the condition again—it's finally `false` because `product` is now 81, which is not less than or equal to `50`, so the loop terminates.

Variable `product` now contains `81`—the first power of 3 greater than 50. The `while` statement's body may be a single statement or a block.

UML Activity Diagram for a `while` Statement

The following UML activity diagram illustrates the preceding `while`'s flow of control:
This diagram introduces the **UML's merge symbol**. The UML represents both the merge symbol and the decision symbol as diamonds. The merge symbol joins two flows of activity into one. In this diagram, the merge symbol joins the transitions from the initial state and from the action state, so they both flow into the decision that determines whether the loop should begin (or continue) executing.

The decision and merge symbols can be distinguished by the number of "incoming" and "outgoing" transition arrows:

- A decision symbol has one transition arrow pointing to the diamond and two or more pointing from it to indicate

possible transitions. Each transition arrow pointing from a decision symbol has a guard condition next to it.

- A merge symbol has two or more transition arrows pointing to the diamond and only one pointing from it to indicate multiple activity flows merging to continue the activity. A merge's transition arrows do not have guard conditions.

The diagram clearly shows the `while` statement's iteration. The transition arrow emerging from the action state points back to the merge, from which program flow transitions back to the decision tested at the beginning of each iteration. The `while` statement executes until the guard condition "product $<=$ 50" becomes false, and thus the guard condition "product $>$ 50" becomes true. Then, the `while` statement reaches its final state, and the program continues with the next statement in sequence.

Generative AI

1 Prompt genAIs to write a Java program that determines the first power of 2 greater than one million. Do they use the techniques we show in the following code? If not, adjust your prompt with hints that help the genAIs generate similar code to that shown here.

Click here to view code image

```
// Power.java
// Calculate the first power of 2 greater than 1,000,000.
public class Power {
   public static void main(String[] args) {
      int product = 2;
      int n = 1;

      while (product <= 1_000_000) {
         product = 2 * product;
         n = n + 1;
      }

      System.out.printf("%s %d%n%s %d%n",
         "First power of 2 greater than 1,000,000 is", product,
         "which is 2 to the power of", n);
   }
}
```

```
First power of 2 greater than 1,000,000 is 1048576
which is 2 to the power of 20
```

3.6 Counter-Controlled Iteration

Let's solve two variations of a problem that averages student grades. Consider the following problem statement:

> A class of ten students took a quiz. The grades (integers in the range 0–100) for this quiz are available to you. Determine the class average on the quiz.

The class average is the sum of the grades divided by the number of students. The program for solving this problem must input each grade, keep track of the total of all grades input, perform the averaging calculation and print the result. We use **counter-controlled iteration** to input the grades one at a time. This technique uses a variable called a **counter** (or **control variable**) to control the number of times statements will execute. Counter-controlled iteration is also called **definite iteration** because the number of iterations is known before the loop executes. In this example, iteration terminates when the counter exceeds 10. This section presents the Java program (Fig. 3.1) to implement the algorithm.

3.6.1 Implementing the Algorithm

In Fig. 3.1, class `ClassAverage`'s `main` method allows the user to enter 10 grades, then calculates and displays the average.

Click here to view code image

```
1  // Fig. 3.1: ClassAverage.java
2  // Solving the class-average problem using counter-controlled iteration.
3  import java.util.Scanner; // program uses class Scanner
4
5  public class ClassAverage {
6     public static void main(String[] args) {
7        // create Scanner to obtain user input
8        Scanner input = new Scanner(System.in);
9
10       // initialization phase
11       int total = 0;  // initialize sum of grades entered by the user
12       int gradeCounter = 1; // initialize grade # to be entered next
13
14       // processing phase uses counter-controlled iteration
15       while (gradeCounter <= 10) { // loop 10 times
16          System.out.print("Enter grade: "); // prompt for next grade
```

```
17            int grade = input.nextInt(); // input the grade
18            total = total + grade; // add grade to total
19            gradeCounter = gradeCounter + 1; // increment counter by 1
20         }
21
22         // termination phase
23         int average = total / 10; // integer division yields integer result
24
25         // display total and average of grades
26         System.out.printf("%nTotal of all 10 grades is %d%n", total);
27         System.out.printf("Class average is %d%n", average);
28      }
29   }
```

```
Enter grade: 67
Enter grade: 78
Enter grade: 89
Enter grade: 67
Enter grade: 87
Enter grade: 98
Enter grade: 93
Enter grade: 85
Enter grade: 82
Enter grade: 100

Total of all 10 grades is 846
Class average is 84
```

Fig. 3.1 | Solving the class-average problem using counter-controlled iteration.

Local Variables in Method `main`

Line 8 declares and initializes `Scanner` variable `input`, which is used to read values entered by the user. Lines 11, 12, 17 and 23 declare the local `int` variables `total`, `gradeCounter`, `grade` and `average`. Variable `grade` stores the user input.

These declarations appear in `main`'s body. Variables declared in a method body are **local variables**. They can be used only from the line of their declaration to the method's closing right brace. A local variable's declaration must appear before the variable is used in that method. A local variable cannot be accessed outside the method in which it's declared. Variable `grade`, declared in the `while` loop's body, can be used only in that block.

Initialization Phase: Initializing Variables `total` and

gradeCounter

The assignments (in lines 11–12) initialize `total` to `0` and `gradeCounter` to `1`. These initializations occur before the variables are used in calculations.

All local variables must be initialized before their values are used in expressions; otherwise, compilation errors occur. Totals are generally initialized to 0. Counters are generally initialized to 0 or 1, depending on how they're used. Throughout the book, we'll show examples of when to use 0 and when to use 1.

Processing Phase: Reading 10 Grades from the User

Line 15 indicates that the `while` statement should continue looping (also called **iterating**) as long as `gradeCounter`'s value is less than or equal to 10. While this condition remains `true`, the `while` statement repeatedly executes the statements in lines 16–19:

- Line 16 displays the prompt `"Enter grade: "`.
- Line 17 reads the grade entered by the user and assigns it to variable `grade`.
- Line 18 adds the new `grade` into the `total`, replacing its previous value.
- Line 19 adds `1` to `gradeCounter` (replacing its previous value) to indicate that the program processed a grade and is ready to input the next grade from the user.

Incrementing `gradeCounter` eventually causes it to exceed 10. Then, the loop terminates because its condition (line 15) becomes `false`.

Termination Phase: Calculating and Displaying the Class Average

When the loop terminates:

- Line 23 calculates the average and assigns the result to the variable `average`.
- Line 26 uses `System.out`'s `printf` method to display the text `"Total of all 10 grades is "` followed by `total`'s value.
- Line 27 uses `printf` to display the text `"Class average is "` followed by the `average`'s value.

When execution reaches line 28, the program terminates.

3.6.2 Integer Division and Truncation

Line 23's averaging calculation produces an integer result. In the sample output, the sum of the grades is 846, which, when divided by 10, should yield 84.6. This is a **floating-point number**—a number with a decimal point. In line 23, however, the result of `total / 10` is the integer 84 because `total` and `10` are both integers. Dividing two integers results in **integer division**—any fractional part of the calculation is **truncated** (i.e., lost). In the next section, we'll see how to obtain a floating-point result from the averaging calculation. Assuming that integer division rounds (rather than truncates) can lead to incorrect results.

3.6.3 Arithmetic Overflow

In Fig. 3.1, line 18 adds each `grade` entered by the user to the `total`:

Click here to view code image

```
total = total + grade; // add grade to total
```

Even this simple statement has a potential problem—adding the integers could result in a value too large to store in an `int` variable. This is known as **arithmetic overflow** and causes undefined behavior, which can lead to unintended results.[4] Figure 2.5's `Addition` program had the same issue in line 17, which calculated the sum of two `int` values entered by the user:

4. "Integer Overflow." Wikipedia, last modified May 18, 2024. https://en.wikipedia.org/wiki/Integer_overflow#Security_ramifications. [Discusses potential problems associated with integer overflow.]

Click here to view code image

```
int sum = number1 + number2; // add numbers, then store total in sum
```

The constants `Integer.MIN_VALUE` and `Integer.MAX_VALUE` represent the minimum and maximum values an `int` variable can store. There are similar constants for the other numeric primitive types. You can find the values of these constants in each class's online documentation. `Integer`'s documentation is located at:

Click here to view code image

```
https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/
   lang/Integer.html
```

3.6.4 Input Validation

Whenever a program receives input from the user, various problems might occur. For example, line 17 of Fig. 3.1

Click here to view code image

```
int grade = input.nextInt(); // input next grade
```

assumes the user will enter an integer grade in the range 0 to 100, but the user could enter:

- an integer less than 0,
- an integer greater than 100,
- an integer outside the range of values that can be stored in an `int` variable,
- a number containing a decimal point or
- a value containing letters or special symbols that do not even represent a number.

Industrial-strength programs must test for all possible erroneous cases to ensure valid inputs. A program that inputs grades should validate them using **range checking** to ensure they are values from 0 to 100. You can then ask the user to re-enter any out-of-range value. If a program requires inputs from a specific set of values (such as non-sequential product codes), you should ensure that each input matches a value in the set.

Generative AI

1 Prompt genAIs with the following code, asking them to fix it so it displays the `count` values 1–4:

Click here to view code image

```
int count = 1;

while (count < 4) {
   count = count + 1;
   System.out.printf("count is: %d%n", count);
}
```

Compare their results to the following code:

Click here to view code image

```
int count = 0;

while (count < 4) {
   count = count + 1;
   System.out.printf("count is: %d%n", count);
}
```

```
count is: 1
count is: 2
count is: 3
count is: 4
```

3.7 Sentinel-Controlled Iteration

Let's generalize Section 3.6's class-average problem. Consider the following problem:

> Develop a class-averaging program that processes grades for an arbitrary number of students each time it is run.

In the previous class-average example, the number of grades (10) was known in advance. For this problem, we do not know how many grades the user will enter during the program's execution. The program must process an arbitrary number of grades.

One way to solve this problem is to use a **sentinel value** (also called a **signal value** or a **flag value**) indicating the end of data entry. After entering all the legitimate grades, the user enters the sentinel value to indicate that no more grades will be entered.

You must choose a sentinel value that cannot be confused with an acceptable input value. Grades on a quiz are nonnegative integers, so –1 is an acceptable sentinel value for this problem. Thus, a run of the class-averaging program might process a stream of inputs such as 95, 96, 75, 74, 89 and –1. The program would then compute and print the class average for the five grades, excluding –1, because it's the sentinel value and should not enter into the averaging calculation.

3.7.1 Implementing Sentinel-Controlled Iteration

Figure 3.2 implements sentinel-controlled iteration. Much of Fig.

3.2 is identical to Fig. 3.1, so we concentrate on the new content here. Although each grade the user enters is an integer, the average calculation will likely produce a floating-point number like 84.6, which an `int` cannot represent. You store floating-point numbers using the types `float` and `double`:

- Variables of the `float` primitive type represent **single-precision floating-point numbers.** They have up to seven significant digits to the right of the decimal point.
- Variables of the `double` primitive type represent **double-precision floating-point numbers**. These require twice as much memory as `float` variables. They have up to 15 significant digits—about twice a `float` variable's precision.

Most programmers use type `double` because all floating-point numbers in a program's source code, such as 7.33 and 0.0975, are represented as type `double` by default. Such values are known as **floating-point literals**. For the precise ranges of Java's floating-point types, visit the Java Language Specification Section 4.2, "Primitive Types and Values."[5]

5. The Java Language Specification, Java SE 22 Edition. "4.2. Primitive Types and Values." Accessed May 21, 2024. https://docs.oracle.com/javase/specs/jls/se22/html/jls-4.html#jls-4.2.

Click here to view code image

```
1  // Fig. 3.2: ClassAverage.java
2  // Solving the class-average problem using sentinel-controlled iteration.
3  import java.util.Scanner; // program uses class Scanner
4
5  public class ClassAverage {
6     public static void main(String[] args) {
7        // create Scanner to obtain user input
8        Scanner input = new Scanner(System.in);
9
10       // initialization phase
11       int total = 0; // initialize grade total
12       int gradeCounter = 0; // initialize # of grades entered so far
13
14       // processing phase
15       // prompt for input and read grade from user
16       System.out.print("Enter grade or -1 to quit: ");
17       int grade = input.nextInt();
18
19       // loop until sentinel value read from user
20       while (grade != -1) {
```

```
21            total = total + grade; // add grade to total
22            gradeCounter = gradeCounter + 1; // increment counter
23
24            // prompt for input and read next grade from user
25            System.out.print("Enter grade or -1 to quit: ");
26            grade = input.nextInt();
27         }
28
29         // termination phase
30         // if user entered at least one grade...
31         if (gradeCounter != 0) {
32            // use a number with decimal point to calculate the average
33            double average = (double) total / gradeCounter;
34
35            // display total and average (with two digits of precision)
36            System.out.printf("%nTotal of the %d grades entered is %d%n",
37               gradeCounter, total);
38            System.out.printf("Class average is %.2f%n", average);
39         }
40         else { // no grades were entered, so output appropriate message
41            System.out.println("No grades were entered");
42         }
43      }
44   }
```

```
Enter grade or -1 to quit: 97
Enter grade or -1 to quit: 88
Enter grade or -1 to quit: 72
Enter grade or -1 to quit: -1

Total of the 3 grades entered is 257
Class average is 85.67
```

Fig. 3.2 | Solving the class-average problem using sentinel-controlled iteration.

Program Logic for Sentinel-Controlled Iteration vs. Counter-Controlled Iteration

Line 12 initializes `gradeCounter` to `0` because no grades have been entered yet. Remember that we're using sentinel-controlled iteration to input the grades, so we increment the variable `gradeCounter` only when the user enters a valid grade. Line 33 declares the `double` variable `average`, which stores the class average as a floating-point number.

Compare this program's logic for sentinel-controlled iteration with Fig. 3.1's counter-controlled iteration:

- In counter-controlled iteration, each loop iteration (lines 15–20 of Fig. 3.1) reads a value from the user for the specified number of iterations.
- In sentinel-controlled iteration, lines 16–17 of Fig. 3.2 prompt for and input the first value before reaching the `while`. This value determines whether the flow of control should enter the `while`'s body. If the condition is `false`, the user entered the sentinel value—no grades were entered, so the `while`'s body does not execute. If the condition is `true`, the body adds the `grade` to the `total` and increments the `gradeCounter`. Then, lines 25–26 prompt for and input the next grade. At this point, program control reaches the loop's closing right brace (line 27). Execution continues by testing the `while`'s condition (line 20), using the most recently entered `grade` to determine whether the loop body should execute again.

Notice that the next `grade` is always input from the user immediately before the program tests the `while` statement's condition. This allows the program to determine whether the value just input is the sentinel value before processing it (i.e., adding it to the `total`). If `grade` contains the sentinel value, the loop terminates without adding –1 to the `total`. In a sentinel-controlled loop, prompts should remind the user of the sentinel.

After the loop terminates, the `if`…`else` statement in lines 31–42 executes. The condition (line 31) determines whether any grades were input. If none were input, the `else` part displays `"No grades were entered"`. After the `if`…`else` executes, the program terminates.

3.7.2 Explicitly and Implicitly Converting Between Primitive Types

If at least one grade was entered, line 33 calculates the average of the grades. Recall from Fig. 3.1 that integer division yields an integer result. Even though variable `average` is declared as a `double`, if we had written the averaging calculation as

Click here to view code image

```
double average = total / gradeCounter;
```

it would lose the fractional part of the quotient before the result is

assigned to `average`. This occurs because `total` and `gradeCounter` are both integers, and integer division yields an integer result.

Most averages are floating-point numbers. To perform a floating-point calculation with integer values, we must temporarily treat these values as floating-point numbers. The **unary cast operator** accomplishes this task. Line 33 of Fig. 3.2 uses the `(double)` cast operator to create a temporary floating-point copy of its operand `total` (which appears to the operator's right). Using a cast operator in this manner is called **explicit conversion** or **type casting**. The value stored in `total` is still an integer.

The calculation now consists of a floating-point value (the temporary `double` copy of `total`) divided by the integer `gradeCounter`. In mixed-type arithmetic expressions, Java performs an operation called **promotion** (or **implicit conversion**) on selected operands. In an arithmetic expression containing `int` and `double` values, copies of the `int` values are promoted to `double` values for use in the expression. In this example, `gradeCounter`'s value is promoted to type `double`, then floating-point division is performed, and the calculation's result is assigned to `average`. Later in this chapter, we discuss all the primitive types. You'll learn more about the promotion rules in Section 5.11.

3.7.3 Formatting Floating-Point Numbers

Lines 36–37 display `gradeCounter`'s and `total`'s values. Line 38 displays the `double` variable `average`'s value. Method `printf` uses the **`%f` format specifier** to format floating-point numbers. By default, `%f` formats `double`s with six digits of precision to the right of the decimal point, but you can specify a **custom precision** between the `%` and the `f`. The average of the three grades in the sample execution was 85.666666…, with the sequence of 6s repeating infinitely. In the format specifier `%.2f`, the `.2` is the custom precision. This rounds the average to the hundredths position, producing `85.67` with `2` digits to the right of the decimal point.

A Note Regarding Floating-Point Number Precision

Floating-point numbers are not always 100% precise, but they have numerous applications. For example, when we speak of a "normal" body temperature of 98.6 °F, we do not need to be precise to many digits. When we read the temperature on a thermometer as 98.6, it may actually be 98.5999473210643. Calling this number simply

98.6 is fine for most applications involving body temperatures.

Floating-point numbers often arise from division, as in this example's class-average calculation. In conventional arithmetic, when we divide 10 by 3, the result is 3.3333333…, with the 3s repeating infinitely. The computer allocates only a fixed amount of space to hold such a value, so the stored floating-point value can be only an approximation.

Due to the imprecise nature of floating-point numbers, type `double` is preferred to `float` because `double` variables can represent floating-point numbers more accurately. So, we primarily use type `double` throughout the book. Assuming floating-point numbers are represented precisely can lead to incorrect results. In some cases, the precision of `float` and `double` variables will be inadequate. For precise floating-point numbers (such as those required by monetary calculations), Java provides class `BigDecimal` (package `java.math`), which we'll discuss in Section 4.11's Objects-Natural Case Study.

Generative AI

1 Prompt genAIs to write a Java program that reads an arbitrary number of integer Fahrenheit temperatures from the user and calculates and displays their floating-point average. Tell the genAIs to assume the user enters temperatures in the range –212 through 212 and to use the sentinel value 999.

Compare their generated code to the following program. If their code uses techniques we have not presented, try to tune the prompt, giving them more details that will help them generate the code using only techniques we've presented so far. For example, tell them to use a `while` loop and to input the first temperature before the loop and each subsequent temperature at the end of the loop's body.

Click here to view code image

```
1  // AverageTemperature.java
2  // Calculate the average of several temperatures
3  import java.util.Scanner;
4
5  public class AverageTemperature {
6     public static void main(String[] args) {
7        Scanner input = new Scanner(System.in);
8
9        int total = 0;
10       int counter = 0;
```

```
11
12          System.out.print("Enter a temperature or 999 to quit: ");
13          int temperature = input.nextInt();
14
15          while (temperature != 999) {
16             total = total + temperature;
17             counter = counter + 1;
18
19             System.out.print("Enter a temperature or 999 to quit: ");
20             temperature = input.nextInt();
21          }
22
23          if (counter != 0) {
24             double average = (double) total / counter;
25             System.out.printf("Average temperature is: %.2f%n", average);
26          }
27          else {
28             System.out.println("No temperatures were entered.");
29          }
30       }
31    }
```

```
Enter a temperature or 999 to quit: 12
Enter a temperature or 999 to quit: -22
Enter a temperature or 999 to quit: 17
Enter a temperature or 999 to quit: 999
Average temperature is: 2.33
```

3.8 Nested Control Statements

We've seen that control statements can be stacked on top of one another (in sequence). In this case study, we examine the only other structured way control statements can be connected—namely, by **nesting** one control statement within another.

3.8.1 Problem Statement

Consider the following problem statement:

> A college offers a course that prepares students for the state licensing exam for real-estate brokers. Last year, ten of the students who completed this course took the exam. The college wants to know how well its students did on the exam. You've been asked to write a program to summarize

the results.

You've been given a list of these 10 students. Next to each name is written a 1 if the student passed the exam or a 2 if the student failed.

Your program should analyze the results of the exam as follows:

- Input each test result (i.e., a 1 or a 2). Display "Enter result" each time the program requests another test result.
- Count the number of test results of each type.
- Display a summary of the test results, indicating the number of students who passed and the number who failed.
- If more than eight students passed, print "Bonus to instructor!"

Problem Statement Observations

After reading the problem statement carefully, we make the following observations:

- The program must process test results for 10 students. A counter-controlled loop can be used because the number of test results is known in advance.
- Each test result is either a 1 or a 2. As the program reads each result, it must determine whether it's a 1 or a 2. We test for a 1 in our algorithm. If the number is not a 1, we assume it's a 2.
- Two counters are used to keep track of the exam results—one to count the number of students who passed the exam and one to count the number who failed.
- After the program has processed all the results, it must decide whether more than eight students passed the exam.

3.8.2 Implementing the Program

Figure 3.3 implements the program with counter-controlled iteration and shows two sample executions. Lines 11–13 and 19 declare the variables we use to process the examination results.

Initializing local variables when they're declared helps you avoid compilation errors from attempts to use uninitialized variables. While Java does not require that local-variable initializations be incorporated into declarations, it does require that local variables have values before they're used in an expression; otherwise, compilation errors occur.

Click here to view code image

```
1  // Fig. 3.3: Analysis.java
2  // Analysis of examination results using nested control statements.
3  import java.util.Scanner; // class uses class Scanner
4
5  public class Analysis {
6     public static void main(String[] args) {
7        // create Scanner to obtain user input
8        Scanner input = new Scanner(System.in);
9
10       // initializing variables in declarations
11       int passes = 0;
12       int failures = 0;
13       int studentCounter = 1;
14
15       // process 10 students using counter-controlled iteration
16       while (studentCounter <= 10) {
17          // prompt user for input and obtain value from user
18          System.out.print("Enter result (1 = pass, 2 = fail): ");
19          int result = input.nextInt();
20
21          // if...else is nested in the while statement
22          if (result == 1) {
23             passes = passes + 1;
24          }
25          else {
26             failures = failures + 1;
27          }
28
29          // increment studentCounter so loop eventually terminates
30          studentCounter = studentCounter + 1;
31       }
32
33       // termination phase; prepare and display results
34       System.out.printf("Passed: %d%nFailed: %d%n", passes, failures);
35
36       // determine whether more than 8 students passed
37       if (passes > 8) {
38          System.out.println("Bonus to instructor!");
39       }
40    }
```

```
41    }
```

```
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 2
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Passed: 9
Failed: 1
Bonus to instructor!
```

```
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 2
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 2
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 2
Enter result (1 = pass, 2 = fail): 2
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Passed: 6
Failed: 4
```

Fig. 3.3 | Analysis of examination results using nested control statements.

The `while` statement (lines 16–31) loops 10 times. During each iteration, it inputs and processes one exam result. The `if...else` statement (lines 22–27) for processing each result is nested in the `while` statement. If the `result` is `1`, the `if...else` statement increments `passes`; otherwise, it assumes the `result` is `2` and increments `failures`. Line 30 increments `studentCounter` before the loop condition is tested again at line 16. After 10 values have been input, the loop terminates, and line 34 displays the number of `passes` and `failures`. The `if` statement at lines 37–39 determines whether more than eight students passed the exam and, if so, outputs the message `"Bonus to instructor!"`.

Figure 3.3 shows the input and output from two sample executions of the program. During the first, the condition at line 37 is `true`—more than eight students passed the exam, so the program

outputs a message to bonus the instructor.

Generative AI

1 Prompt genAIs to simplify the following program by replacing the nested `if...else` statement with a single statement that uses a conditional expression. Execute the code they return to ensure it produces the same results.

Click here to view code image

```
1  // OddOrEven.java
2  // Determine whether each integer is odd or even.
3  import java.util.Scanner;
4
5  public class OddOrEven {
6     public static void main(String[] args) {
7        Scanner input = new Scanner(System.in);
8        int counter = 1;
9
10       while (counter <= 3) {
11          System.out.print("Enter an integer: ");
12          int number = input.nextInt();
13
14          if (number % 2 == 0) {
15             System.out.printf("%d is even%n", number);
16          }
17          else {
18             System.out.printf("%d is odd%n", number);
19          }
20
21          counter = counter + 1;
22       }
23    }
24 }
```

```
Enter an integer: 2
2 is even
Enter an integer: 3
3 is odd
Enter an integer: 5
5 is odd
```

3.9 Compound Assignment Operators

The **compound assignment operators** enable you to abbreviate assignment expressions. You can abbreviate the statement

```
c = c + 3; // adds 3 to c
```

with the **addition compound assignment operator**, `+=`, as

Click here to view code image

```
c += 3; // adds 3 to c more concisely
```

The `+=` operator adds its right operand's value to the left-side variable's value then stores the result in the left-side variable. Thus, the assignment expression `c+=3` adds `3` to `c`. Compound assignment operators perform an implicit conversion on the right-hand expression's value to match the type of the variable on the operator's left side. So, if the left-side variable is an `int` and the right-side expression results in a `double`, the `double` value's floating-point part will be truncated.[6] The following table shows the arithmetic compound assignment operators, sample expressions and explanations of what the operators do:

Compound operator	Sample expression	Explanation
Assume: `int c = 3, d = 5, e = 4, f = 6, g = 12;`		
`+=`	`c += 7`	`c = c + 7`
`-=`	`d -= 4`	`d = d - 4`
`*=`	`e *= 5`	`e = e * 5`
`/=`	`f /= 3`	`f = f / 3`
`%=`	`g %= 9`	`g = g % 9`

6. The Java Language Specification, Java SE 22 Edition. Section 15.26.2, "Compound Assignment Operators." Accessed August 28, 2024. https://docs.oracle.com/javase/specs/jls/se22/html/jls-15.html#jls-15.26.2.

Generative AI

1 Prompt genAIs with the following code, asking what the value of `y` is after the code executes:

```
int y = 5;
y += 3.3;
```

Confirm that the genAIs indicate `y` is 8. You might expect a compilation error from the assignment statement. Compound

assignment operators implicitly convert the right-hand expression's value to match the type of the variable on the operator's left side. So, 3.3 is truncated to 3.

3.10 Increment and Decrement Operators

The following table summarizes the unary operators for adding 1 to or subtracting 1 from the value of a numeric variable—these are the unary **increment operator**, `++`, and the unary **decrement operator**, `--`. Unlike binary operators, the unary increment and decrement operators, by convention, should be placed next to their operands with no intervening spaces:

Operator	Expression	Description
Prefix increment	`++number`	Increment `number` by `1`, then use the new value of `number` in the expression in which `number` resides.
Postfix increment	`number++`	Use the current value of `number` in the expression in which `number` resides, then increment `number` by `1`.
Prefix decrement	`--number`	Decrement `number` by `1`, then use the new value of `number` in the expression in which `number` resides.
Postfix decrement	`number--`	Use the current value of `number` in the expression in which `number` resides, then decrement `number` by `1`.

Prefix Increment vs. Postfix Increment

Figure 3.4 demonstrates the difference between the prefix increment and postfix increment versions of the `++` increment operator. The decrement operator (`--`) works similarly.

Click here to view code image

```
// Fig. 3.4: Increment.java
// Prefix increment and postfix increment operators.

public class Increment {
      public static void main(String[] args) {
         // demonstrate postfix increment operator
         int c = 5;
         System.out.printf("c before postincrement: %d%n", c); // prints 5
         System.out.printf("    postincrementing c: %d%n", c++); // prints
         System.out.printf(" c after postincrement: %d%n", c); // prints 6
```

```
11
12            System.out.println(); // skip a line
13
14            // demonstrate prefix increment operator
15            c = 5;
16            System.out.printf(" c before preincrement: %d%n", c); // prints 5
17            System.out.printf("     preincrementing c: %d%n", ++c); // prints
18            System.out.printf("  c after preincrement: %d%n", c); // prints 6
19        }
20    }
```

```
c before postincrement: 5
    postincrementing c: 5
 c after postincrement: 6

 c before preincrement: 5
     preincrementing c: 6
  c after preincrement: 6
```

Fig. 3.4 | Prefix increment and postfix increment operators.

Line 7 initializes the variable `c` to `5`, and line 8 outputs `c`'s initial value. Line 9 outputs the value of the expression `c++`. This expression postincrements the variable `c`, so the statement outputs `c`'s original value (`5`) again, then `c`'s value is incremented (to 6). Line 10 outputs `c`'s new value (`6`) to prove that the variable's value was indeed incremented in line 9.

Line 15 resets `c`'s value to `5`, and line 16 outputs `c`'s value. Line 17 outputs the value of the expression `++c`, which preincrements `c`, so the statement outputs `c`'s new value (`6`). Line 18 outputs `c`'s value again to show that the value of `c` is still `6` after line 17 executes.

Simplifying Increment Statements

The arithmetic compound assignment operators and the increment and decrement operators can be used to simplify program statements. For example, the three assignment statements in Fig. 3.3 (lines 23, 26 and 30)

Click here to view code image

```
passes = passes + 1;
failures = failures + 1;
studentCounter = studentCounter + 1;
```

can be written more concisely with compound assignment operators

as

```
passes += 1;
failures += 1;
studentCounter += 1;
```

with prefix increment operators as

```
++passes;
++failures;
++studentCounter;
```

or with postfix increment operators as

```
passes++;
failures++;
studentCounter++;
```

When incrementing or decrementing a variable in a statement by itself, the prefix increment and postfix increment forms have the same effect, and the prefix decrement and postfix decrement forms have the same effect. It's only when a variable appears in the context of a larger expression that preincrementing and postincrementing the variable have different effects (and similarly for predecrementing and postdecrementing).

Generative AI

1 Prompt genAIs to explain what the following code does.

Click here to view code image

```
1  Mystery.java
2  public class Mystery {
3     public static void main(String[] args) {
4        int x = 1;
5        int total = 0;
6
7        while (x <= 10) {
8           int y = x * x;
9           System.out.println(y);
10          total += y;
11          ++x;
12       }
13
14       System.out.printf("Total is %d%n", total);
15    }
```

Compare their answers to the following: In a vertical column, the code prints the squares of 1–10 then prints the sum of those squares.

```
1
4
9
16
25
36
49
64
81
100
Total is 385
```

3.11 Primitive Types

Java requires all variables to have a type.[7] In C and C++, programmers frequently have to write separate versions of programs to support different computer platforms because the primitive types are not guaranteed to be identical among computers. For example, an `int` on one machine might be represented by 16 bits (2 bytes) of memory, on another machine by 32 bits (4 bytes), and on another by 64 bits (8 bytes). In Java, `int` values are always 32 bits (4 bytes). The eight primitive types in Java—`boolean`, `byte`, `char`, `double`, `float`, `int`, `long` and `short`—are portable across all computer platforms that support Java. For a summary of these types' value ranges, visit the Java Language Specification Section 4.2, Primitive Types and Values.[8] Because the designers of Java want to ensure portability, they use internationally recognized standards for character formats (Unicode; `https://unicode.org`) and floating-point numbers (IEEE 754; `https://standards.ieee.org/ieee/754/6210/`).

7. We'll see an exception to this with lambdas in Chapter 14, Functional Programming with Lambdas & Streams.

8. The Java Language Specification, Java SE 22 Edition. Section 4.2, "Primitive Types and Values." Accessed June 1, 2024. `https://docs.oracle.com/javase/specs/jls/se22/html/jls-4.html#jls-4.2`.

Generative AI

1 Prompt genAIs to write a Java program that shows the minimum and maximum values of Java's integer primitive types, then run the generated program.

3.12 Objects-Natural Case Study: Super-Sized Integers

The explosive growth of Internet communications and data storage on Internet-connected devices has dramatically increased the importance of privacy and security. Cryptography has been used for thousands of years to encode data, making it difficult (and hopefully impossible) for unauthorized users to read,[9,10] which is critically important in today's connected world. Most websites (including `deitel.com`) use HTTPS (HyperText Transfer Protocol Secure) to encrypt and decrypt your Internet interactions.

9. "Cryptography." Accessed June 1, 2024. https://en.wikipedia.org/wiki/Cryptography#History.
10. Binance Academy, "History of Cryptography." Accessed June 1, 2024. https://www.binance.vision/security/history-of-cryptography.

Some Applications Need Integers Outside a `long` Integer's Range

Java's primitive type `long` can store values in the range –9,223,372,036,854,775,808 to 9,223,372,036,854,775,807—a maximum of 19 decimal digits, which can represent integers in the low quintillions. Many cryptography algorithms perform calculations with integers far larger than we can store in a `long`.

One of the most popular encryption/decryption schemes is **RSA Public-Key Cryptography**, commonly used to secure data transmitted over the Internet.[11,12,13] Industrial-strength RSA implementations work with enormous prime numbers consisting of hundreds of digits, far more than primitive type `long` can hold.

11. "RSA (Cryptosystem)." Accessed June 1, 2024. https://en.wikipedia.org/wiki/RSA_(cryptosystem).
12. "RSA Algorithm." Accessed June 1, 2024. https://

simple.wikipedia.org/wiki/RSA_algorithm.

13. K. Moriarty, B. Kaliski, J. Jonsson and A. Rusch, "PKCS #1: RSA Cryptography Specifications Version 2.2," November 2016. Accessed June 1, 2024. https://tools.ietf.org/html/rfc8017.

A key reason RSA is so secure is the sheer amount of time required to factor the product of those primes, even for today's most powerful supercomputers. Public-key cryptography is also used to secure the blockchain technology behind cryptocurrencies like Bitcoin.[14] Later in the book, we'll discuss public-key cryptography.

14. Sarah Rothrie, "How Blockchain Cryptography Is Fighting the Rise of Quantum Machines," December 28, 2018. Accessed June 1, 2024. https://coincentral.com/blockchain-cryptography-quantum-machines/.

Integers Outside `long`'s Range Require Custom Programming

So, how can we manipulate huge integers with arbitrary numbers of digits? Applications requiring integers outside `long`'s range require custom programming. We could use the Java class mechanism (mentioned in Section 1.4 and covered in detail in Chapter 8) to develop a custom class that handles huge-integer arithmetic—but that could take months of effort. The objects-natural approach encourages us to check first for classes in the Java API or in free, open-source class libraries to determine if such a "huge-integer" class already exists.

Huge Integers with the Java API's `java.math.BigInteger` Class

There are preexisting classes for creating and manipulating huge integers. For this example, we'll demonstrate the **`BigInteger` class** from the Java API's `java.math` package:

Click here to view code image

```
https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/
   math/BigInteger.html
```

With the objects-natural approach in Fig. 3.5, we'll use objects of the `BigInteger` class without having to understand how that class is implemented.[15,16] We `import` it (line 4), use it to create `BigInteger` objects (lines 12–15), then perform calculations with them and display the results (lines 23–30).

15. If you drive a car, you use it effectively without having to understand how car engines, steering, brakes and transmission systems work internally. After you get deeper into Java, you might want to study `BigInteger`'s source code (approximately 5,000 lines, including extensive Javadoc comments) to learn how it's implemented. In our object-oriented programming presentation later in this book, you'll learn techniques you could use to create your own huge-integer class.

16. You might say, "I insist on knowing how everything I use in Java is implemented." Reflect for a moment on the fact that you're using a Java compiler to translate your Java programs into the bytecodes that the Java Virtual Machine then translates into your computer's machine language. Do you know how the compiler is implemented? Are you sure that it correctly translates each of your Java programs so they'll run properly and produce correct results?

Click here to view code image

```
1  // Fig. 3.5: UsingBigInteger.java
2  // Creating and manipulating super-sized integers
3  // with objects of the Java API's BigInteger class.
4  import java.math.BigInteger;
5
6  public class UsingBigInteger {
7     public static void main(String[] args) {
8        // display the largest value a long can store
9        System.out.printf("LARGEST LONG VALUE: %d%n%n", Long.MAX_VALUE);
10
11       // can create BigIntegers from Strings or integers
12       BigInteger value1 =
13          new BigInteger("10000000000000000000000000000"); // 30 digits
14       BigInteger value2 = BigInteger.valueOf(9_223_372_036_854_775_807L);
15       BigInteger value3 = BigInteger.valueOf(17);
16
17       System.out.println("INITIAL VALUES");
18       System.out.printf("BigInteger value1: %s%n", value1);
19       System.out.printf("BigInteger value2: %s%n", value2);
20       System.out.printf("BigInteger value3: %s%n", value3);
21
22       System.out.println("\nADD, SUBTRACT AND MULTIPLY BIGINTEGERS");
23       System.out.printf("     value1.add(value2): %s%n",
24          value1.add(value2));
25       System.out.printf("value1.subtract(value2): %s%n",
26          value1.subtract(value2));
27       System.out.printf("value1.multiply(value2): %s%n",
28          value1.multiply(value2));
29       System.out.printf("value1.multiply(value3): %s%n",
30          value1.multiply(value3));
31    }
32 }
```

```
LARGEST TYPE LONG VALUE: 9223372036854775807

INITIAL VALUES
BigInteger value1: 10000000000000000000000000000
BigInteger value2: 9223372036854775807
BigInteger value3: 17

ADD, SUBTRACT AND MULTIPLY BIGINTEGERS
     value1.add(value2): 10000000000922337203685477580 7
value1.subtract(value2): 99999999990776627963145224193
value1.multiply(value2): 92233720368547758070000000000000000000000000000000
value1.multiply(value3): 170000000000000000000000000000
```

Fig. 3.5 | Creating and manipulating super-sized integers with objects of the Java API's `BigInteger` class.

Largest Value of a `long`

Line 9 displays **`Long.MAX_VALUE`**—a constant representing the maximum value a `long` variable can store (9,223,372,036,854,775,807). As you'll see, each `BigInteger` calculation in this example results in a value greater than `Long.MAX_VALUE`. Each primitive type (e.g., `long`) has a corresponding class (e.g., the **`Long` class**) in the `java.lang` package. These classes provide the constants `MIN_VALUE` and `MAX_VALUE`, representing the smallest and largest values the corresponding primitive types can store.

Initializing a `BigInteger` with a String

If a `BigInteger`'s initial value exceeds the maximum integer value Java supports, you can initialize a new `BigInteger` object with a string representing the integer value, as we did with the 30-digit integer in lines 12–13.

Initializing a `BigInteger` with an Integer

Lines 14–15 ask the `BigInteger` class to create new `BigInteger` objects and initialize them with primitive-type integer values. Classes provide methods to perform tasks. Each method we've used so far was called on an object. For example, to display text, Fig. 3.5 calls the `printf` and `println` methods on the `System.out` object. Some classes provide methods you call directly on the class rather than on an object of the class. These are called **`static`** methods. You call a `static` method by specifying the class name followed by

a dot (`.`) separator and the method name, as in

ClassName.methodName(*arguments*)

Class `BigDecimal`'s `static` `valueOf` **method** creates and returns a new `BigInteger` object initialized with `valueOf`'s primitive-type integer argument:

- Line 14 initializes `value2` with a `BigInteger` containing a `long`'s maximum value. The `L` ("el") suffix at the end of `9_223_372_036_854_775_807L` indicates that the value is a `long` integer. Without the `L`, the Java compiler treats whole numbers in code as `int` values—in this case, line 14 would cause a compilation error because this value is much larger than an `int`'s maximum positive value (2,147,483,647).[17]

  17. We could have used `Long.MAX_VALUE` in line 14 rather than `9_223_372_036_854_775_807L`.

- Line 15 initializes `value3` with a `BigInteger` containing the `int` value 17.

Lines 18–20 display the initial values of the `BigInteger`s `value1`, `value2` and `value3`.

How Does `printf`'s String Specifier `%s` Understand a `BigInteger` Object?

Every Java primitive-type value and object has a string representation. When you format any value or object using `printf` with the `%s` specifier, Java converts that value or object to a string. For an object, Java does this by calling the object's `toString` method, which we'll discuss in later chapters. The `toString` method is also called when you display an object with `System.out`'s `print` and `println` methods.

Performing `BigInteger` Arithmetic

Class `BigInteger` provides various methods for common arithmetic operations. Lines 23–30 demonstrate the `BigInteger` methods `add`, `subtract` and `multiply`. `BigInteger` correctly handles integers of any number of digits. Each result is a value with more digits than can be represented by Java's integer primitive types. The multiplication in line 28, for example, produces a 48-digit integer. Though a `BigInteger` can represent any integer value, it does not use the primitive types understood by the Java virtual machine. So

calculations with `BigInteger` run (possibly much) slower than those with primitive types.[18]

18. For more details on `BigInteger` performance, refer to the API Note in `BigInteger`'s documentation. "Class `BigInteger`." Accessed September 21, 2024. https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/math/BigInteger.html.

Generative AI

1 Prompt genAIs to write a Java program that calculates and displays the cube of 100,000,000,000,000 (100 trillion). Execute the generated code and compare the code and output to the following:

Click here to view code image

```
BigInteger value = new BigInteger("100000000000000");
System.out.printf("      value is: %s%nvalue cubed is: %s%n",
   value, value.multiply(value.multiply(value)));
```

```
      value is: 100000000000000
value cubed is: 1000000000000000000000000000000000000000000000
```

3.13 Wrap-Up

In this chapter, we demonstrated the `if` single-selection statement, the `if`...`else` double-selection statement and the `while` iteration statement. We computed the average of a 10 student grades with counter-controlled iteration. Then, we calculated the average of an arbitrary set student grades using sentinel-controlled iteration. We introduced the compound assignment operators and the increment and decrement operators. We also discussed Java's eight primitive types, noting that they're portable across platforms.

In our Objects-Natural Case Study, we created and manipulated `BigInteger` objects to perform integer arithmetic with values outside the range supported by Java's integer primitive types. Throughout the chapter, you used generative AI to leverage your Java learning experience. In Chapter 4, we continue our control-statements discussion, introducing the `for`, `do`...`while` and `switch` statements. We also introduce the logical operators for creating compound conditions.

Special Section: Generative AI Exercises

For each of the following questions, try multiple genAIs to determine which performs best. Always verify the results. Keep in mind that they sometimes make mistakes and even hallucinate, creating unreliable information.

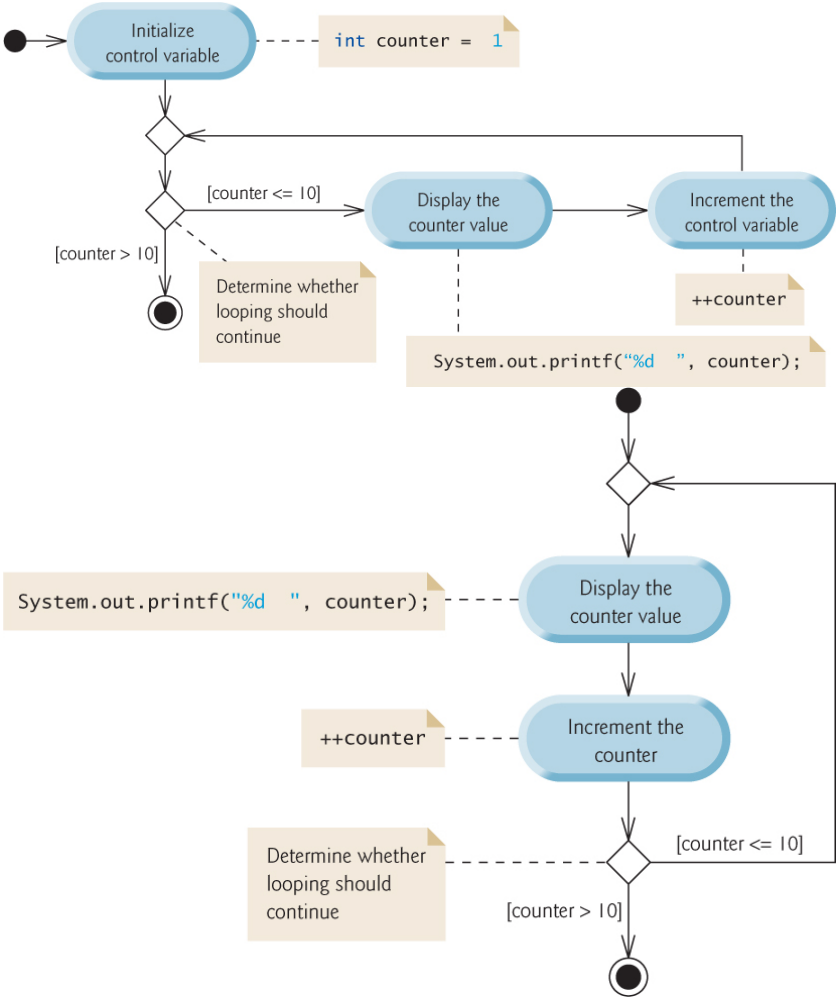
3.1 ***(Generative AI: Control Statements Exercises)*** Prompt genAIs to suggest introductory exercises with answers for practice with Java's `if` and `while` control statements. Limit the suggestions to exercises that can be completed in a maximum of 50 lines of code.

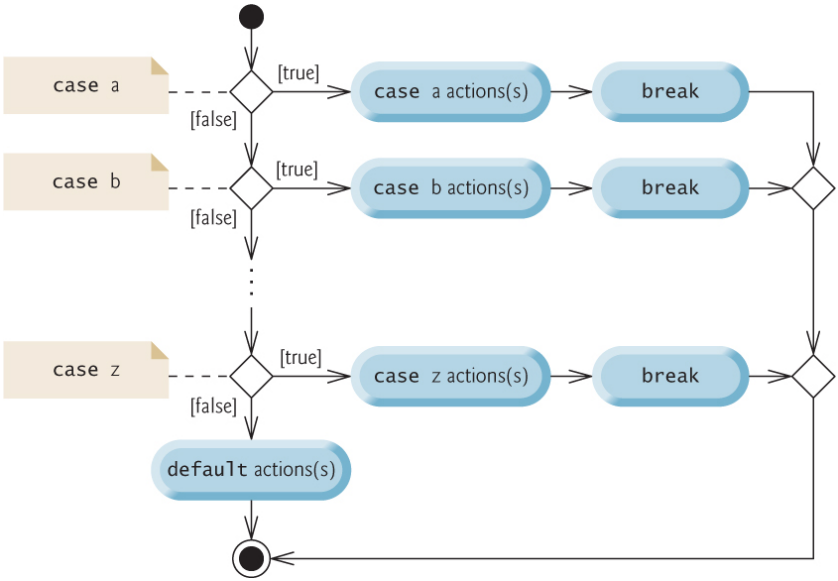
3.2 ***(Generative AI: Filling in Missing Code)*** Remove some code from a program, then tell genAIs what the program is supposed to do and ask them to fill in the missing code.

3.3 ***(Generative AI: Super-Sized Integers)*** Prompt genAIs to write a tutorial on applications of `BigInteger`s.

```
for (int counter = 1; counter <= 10; ++counter)
```

- `for` keyword
- Control variable
- Required semicolon
- Required semicolon
- Initial value of control variable
- Loop-continuation condition
- Incrementing the control variable


Initialize control variable
int counter = 1
[counter <= 10]
Display the counter value
Increment the control variable
[counter > 10]
Determine whether looping should continue
++counter
System.out.printf("%d ", counter);
System.out.printf("%d ", counter);
Display the counter value
++counter
Increment the counter
Determine whether looping should continue
[counter <= 10]
[counter > 10]



Control Statements: Part 2

Objectives

In this chapter, you'll:

- Use the `for` and `do...while` iteration statements to execute statements in a program repeatedly.
- Perform multiple selection using the `switch` selection statement.

- Use the `break` and `continue` statements to alter the flow of control.
- Use the logical operators to form compound conditional expressions that test multiple conditions in control statements.
- Use generative AI to leverage your Java learning experience.
- Continue our objects-natural approach with a case study on using objects of the Java API's `BigDecimal` class to perform precise monetary calculations.

Outline

4.1 Introduction

4.2 Essentials of Counter-Controlled Iteration

4.3 `for` Iteration Statement

4.4 Examples Using the `for` Statement

4.5 Summing the Even Integers from 2 to 20 with the `for` Statement

4.6 Calculating Compound Interest with the `for` Statement

4.7 `do...while` Iteration Statement

4.8 `switch` Multiple-Selection Statement

4.9 `break` and `continue` Statements

4.10 Logical Operators

4.10.1 Conditional AND (`&&`) Operator

4.10.2 Conditional OR (`||`) Operator

4.10.3 Short-Circuit Evaluation of Complex Conditions

4.10.4 Boolean Logical AND (`&`) and Inclusive OR (`|`) Operators

4.10.5 Boolean Logical Exclusive OR (`^`) Operator

4.10.6 Logical Complement (`!`) Operator

4.10.7 Example: Generating Logical-Operator Truth Tables

4.11 Objects-Natural Case Study: Precise Monetary Calculations with Java API Class `BigDecimal`

4.12 Wrap-Up

Special Section: Generative AI Exercises

4.1 Introduction

This chapter introduces all but one of Java's remaining control statements. We demonstrate the `for`, `do...while` and `switch` statements. We explore the essentials of counter-controlled iteration using `while` and `for`. We use a `switch` statement to count the number of A, B, C, D and F grade equivalents in a set of numeric grades entered by the user. We introduce the `break` and `continue` program-control statements. We discuss the logical operators, which enable you to use more complex conditional expressions in control statements. Throughout the chapter, you'll use generative AI to leverage your Java learning experience. We discuss the challenges of processing monetary amounts with floating-point types like `double`. Then, in Section 4.11's Objects-Natural Case Study, we represent and manipulate monetary amounts precisely using objects of the Java API class `BigDecimal`.

4.2 Essentials of Counter-Controlled Iteration

Let's use the `while` iteration statement (Section 3.5) to formalize the elements of counter-controlled iteration:

- a control variable (or loop counter),
- the **initial value** of the control variable,
- the **increment** by which the control variable is modified each time through the loop (also known as each **iteration of the loop**), and
- the **loop-continuation condition** that determines if looping should continue.

Consider the program of Fig. 4.1, which uses a loop to display 1–10. Lines 6, 8 and 10 define the elements of counter-controlled iteration. Line 6 declares the control variable (`counter`) as an `int`, reserves space for it in memory and sets its initial value to `1`. The `while` statement's loop-continuation condition (line 8) tests whether the control variable's value is less than or equal to `10` (the final value for which the condition is `true`). The program performs this `while` statement's body even when the control variable is `10`. The loop terminates when the control variable exceeds `10` (that is, `counter` becomes `11`). Line 9 displays `counter`'s value during each

iteration. Line 10 increments the control variable by 1 at the end of each iteration to prepare to process the next value.

Click here to view code image

```
1  // Fig. 4.1: WhileCounter.java
2  // Counter-controlled iteration with the while iteration statement.
3
4  public class WhileCounter {
5     public static void main(String[] args) {
6        int counter = 1; // declare and initialize control variable
7
8        while (counter <= 10) { // loop-continuation condition
9           System.out.printf("%d  ", counter);
10          ++counter; // increment control variable
11       }
12
13       System.out.println();
14    }
15 }
```

```
1 2 3 4 5 6 7 8 9 10
```

Fig. 4.1 | Counter-controlled iteration with the `while` iteration statement.

You should always control counting loops with integers. Recall that floating-point values are approximate. So, controlling counting loops with floating-point variables can result in imprecise counter values and inaccurate termination tests that could cause an infinite loop.

Generative AI

1 Prompt genAIs, asking them to find and correct the error in the following code, which should print the values 1–5:

```
int n = 1;

while (n < 5) {
   System.out.println(n++);
}
```

Confirm that the genAIs point out the loop currently prints the values 1–4 and, to print 1–5, the loop-continuation condition should use `<=` rather than `<` or use the final value `6` rather than `5`.

4.3 `for` Iteration Statement

The **`for` iteration statement** specifies the counter-controlled-iteration details in a single line of code. Typically, you use `for` statements for counter-controlled iteration and `while` statements for sentinel-controlled iteration, but each can be used for either iteration type. Figure 4.2 reimplements the program of Fig. 4.1 using a `for` statement.

Click here to view code image

```
1  // Fig. 4.2: ForCounter.java
2  // Counter-controlled iteration with the for iteration statement.
3
4  public class ForCounter {
5     public static void main(String[] args) {
6        // for statement header includes initialization, loop-continuation
7        // condition and increment separated by required semicolons
8        for (int counter = 1; counter <= 10; ++counter) {
9           System.out.printf("%d  ", counter);
10       }
11
12       System.out.println();
13    }
14 }
```

```
1 2 3 4 5 6 7 8 9 10
```

Fig. 4.2 | Counter-controlled iteration with the `for` iteration statement.

The `for` statement (lines 8–10) declares the control variable `counter` and initializes it to `1`. Next, the statement tests the loop-continuation condition (`counter <= 10`) between the two required semicolons. The condition is `true` because `counter`'s initial value is `1`, so line 9 displays `counter`'s value (`1`). Next, `++counter` increments `counter` by 1. Then, the loop-continuation condition is tested again to determine whether to proceed with the loop's next iteration. At this point, `counter`'s value is `2`, so the condition is still `true`, and the program executes line 9. This process continues until the loop displays 1–10. When `counter`'s value becomes `11`, the loop-continuation condition is `false`, so iteration terminates, and the program continues with the first statement after the loop (line

12).

Off-By-One Errors

If you incorrectly specified the loop-continuation condition in Fig. 4.2 as `counter < 10`, the loop would iterate only nine times—a common logic error called an **off-by-one error**. An incorrect final value in a loop-continuation condition can also cause an off-by-one error. In a loop that counts from 1 through 10, the condition should be `counter <= 10` or `counter < 11`. Many programmers prefer zero-based counting in which to count 10 times, `counter` would be initialized to 0, and the loop-continuation test would be `counter < 10`.[1]

1. In Chapter 6, we'll see examples of zero-based counting.

A Closer Look at the `for` Statement's Header

Let's take a closer look at the `for` statement in Fig. 4.2:
This line—including the keyword `for` and everything in the parentheses after `for` (line 8 in Fig. 4.2)—is sometimes called the **`for` statement header**.

General Format of a `for` Statement

The general format of the `for` statement is

Click here to view code image

```
for (initialization; loopContinuationCondition; increment) {
   statement(s)
}
```

where

- *initialization* declares the loop's control variable and provides its initial value,
- *loopContinuationCondition*—between the two required semicolons—determines whether the loop should continue executing, and
- *increment* modifies the control variable's value after the loop body executes, so the loop-continuation condition eventually becomes `false`.

If the *loopContinuationCondition* is initially `false`, the `for`

statement's body does not execute—execution continues after the `for` with the next statement in sequence. The *initialization* and *increment* expressions can be comma-separated lists of initialization expressions or increment expressions—the expressions in each evaluate left to right.

Placing a semicolon immediately after the right parenthesis of a `for` header makes that `for`'s body an empty statement. This usually is a logic error.

Infinite loops occur when an iteration statement's *loopContinuationCondition* never becomes `false`. To prevent this in a counter-controlled loop, modify the control variable during each loop iteration so the loop-continuation condition eventually becomes `false`.

Scope of a `for` Statement's Control Variable

If the *initialization* expression declares the control variable, it can be used only in that `for` statement—not beyond it. This restricted use is the variable's **scope**, which defines its lifetime and where it can be used in a program. For example, a variable's scope is from its declaration point in a block to the right brace that closes the block. As you'll see in Chapter 5, Methods, it's good practice to limit each variable's scope to only where it's needed.

Expressions in a `for` Statement's Header Are Optional

A `for` header's three expressions are optional:

- If you omit the *loopContinuationCondition*, the condition is always `true`, creating an infinite loop. We'll see later that this is sometimes useful.
- You might omit the *initialization* expression if the program initializes the control variable before the loop.
- You might omit the *increment* expression if the program calculates the increment in the loop's body or if no increment is needed.

The *increment* expression in a `for` acts like a standalone statement at the end of the `for`'s body. Therefore, the following *increment* expressions are equivalent in a `for`:

```
counter = counter + 1
```

```
counter += 1
++counter
counter++
```

The increment expression does not appear in a larger expression, so preincrementing and postincrementing have the same effect. We prefer preincrement.[2] The increment may also be negative, in which case it's a **decrement**, and the loop counts downward.

2. This preincrement preference comes from our experience with C++, where preincrement sometimes performs better than postincrement. In Java, for a standalone counter increment, such as in a `for`'s increment expression, there's no performance advantage of preincrement over postincrement, so the compiler produces the same code. Many Java programmers prefer postincrement. Be careful when using preincrement and postincrement in more complex expressions where they operate differently.

Using a `for` Statement's Control Variable in the Statement's Body

Programs frequently display the control variable's value or use it in calculations in the `for` loop's body, but this is not required. You can change the control variable's value in the loop's body, but this can lead to subtle errors. If you must modify the control variable's value in the loop's body, prefer `while` to `for`.

UML Activity Diagram for the `for` Statement

The following UML activity diagram of Fig. 4.2's `for` statement shows that initialization occurs once before the statement tests the loop-continuation condition the first time. Incrementing occurs after the body statement executes:

Generative AI

1 Prompt genAIs with Fig. 4.2's code and ask them to explain the `for` loop's execution step-by-step.

4.4 Examples Using the `for` Statement

The following examples show techniques for varying the control variable in a `for` statement. In each case, we write only the appropriate `for` header. Note the change in the relational operator

for the loops that decrement the control variable.

a. Vary the control variable from `1` to `100` in increments of `1`.

Click here to view code image

```
for (int i = 1; i <= 100; ++i)
```

b. Vary the control variable from `100` to `1` in decrements of `1`.

Click here to view code image

```
for (int i = 100; i >= 1; --i)
```

c. Vary the control variable from `7` to `77` in increments of `7`.

Click here to view code image

```
for (int i = 7; i <= 77; i += 7)
```

d. Vary the control variable from `20` to `2` in decrements of `2`.

Click here to view code image

```
for (int i = 20; i >= 2; i -= 2)
```

e. Vary the control variable over the values `2`, `5`, `8`, `11`, `14`, `17`, `20`.

Click here to view code image

```
for (int i = 2; i <= 20; i += 3)
```

f. Vary the control variable over the values `99`, `88`, `77`, `66`, `55`, `44`, `33`, `22`, `11`, `0`.

Click here to view code image

```
for (int i = 99; i >= 0; i -= 11)
```

Do not use equality operators (`!=` or `==`) in a loop-continuation condition if the loop's control variable increments or decrements by more than `1`. For example, in the following `for` header, `counter != 10` never becomes `false`, resulting in an infinite loop because `counter` increments by 2 after each iteration, producing the odd values 3, 5, 7, 9, 11, …:

Click here to view code image

```
for (int counter = 1; counter != 10; counter += 2)
```

A common logic error in a loop that counts downward is using an incorrect relational operator in the loop-continuation condition—for

example, `i<= 1` instead of `i>= 1` in a loop counting down to `1`. Counting loops are error-prone. In subsequent chapters, we'll introduce lambdas and streams technologies that you can use to eliminate such errors.

Generative AI

1 Prompt genAIs, asking them to find and correct the error(s) in the following code segment, which should display the odd integers from 19 to 1:

Click here to view code image

```
for (int i = 19; i >= 1; i += 2) {
   System.out.println(i);
}
```

Confirm that the genAIs replace the `+=` operator in the `for` header's increment section with `-=`.

4.5 Summing the Even Integers from 2 to 20 with the `for` Statement

Figure 4.3 uses a `for` statement to sum the even integers from 2 to 20 and store the result in the `int` variable named `total`. When lines 9–11 execute, each loop iteration adds the control variable `number`'s value to `total`'s current value and stores the result in `total`.

Click here to view code image

```
1  // Fig. 4.3: Sum.java
2  // Summing integers with the for statement.
3
4  public class Sum {
5     public static void main(String[] args) {
6        int total = 0;
7
8        // total even integers from 2 through 20
9        for (int number = 2; number <= 20; number += 2) {
10          total += number;
11       }
12
13       System.out.printf("Sum is %d%n", total);
14    }
```

```
15 }
```

```
Sum is 110
```

Fig. 4.3 | Summing integers with the `for` statement.

Generative AI

1 Prompt genAIs, asking them to write code that uses a `for` statement and the `int` variables `sum` and `count` to sum the odd integers from 1 through 99, then displays the result. Also, ask them what the sum should be. Compare the generated code to the following and ensure it produces the correct sum:

Click here to view code image

```
int sum = 0;

for (int count = 1; count <= 99; count += 2) {
   sum += count;
}

System.out.printf("Sum is: %d%n", sum);
```

```
Sum is: 2500
```

4.6 Calculating Compound Interest with the `for` Statement

Let's use a `for` statement to calculate compound interest. Consider the following problem:

> A person invests $1,000 in a savings account yielding 5% interest. Assuming all the interest is left on deposit, calculate and print the amount of money in the account at the end of each year for 10 years. Use the following formula to determine the amounts:
>
> $$a = p\,(1 + r)^n$$

where

p is the original principal amount invested
r is the annual interest rate (e.g., use 0.05 for 5%)
n is the number of years
a is the amount on deposit at the end of the *n*th year.

Figure 4.4 uses a loop to perform the compound-interest calculation for each of the 10 years the money remains on deposit. Lines 6, 7 and 15 declare the `double` variables `principal`, `rate` and `amount`. Lines 6–7 initialize `principal` to `1000.00` and `rate` to `0.05` (5%). Line 15 initializes `amount` to the compound-interest calculation's result.

Click here to view code image

```
1  // Fig. 4.4: Interest.java
2  // Compound-interest calculations with for.
3
4  public class Interest {
5     public static void main(String[] args) {
6        double principal = 1000.00; // initial amount before interest
7        double rate = 0.05; // interest rate
8
9        // display headers
10       System.out.printf("%s%20s%n", "Year", "Amount on deposit");
11
12       // calculate amount on deposit for each of ten years
13       for (int year = 1; year <= 10; ++year) {
14          // calculate new amount on deposit for specified year
15          double amount = principal * Math.pow(1.0 + rate, year);
16
17          // display the year and the amount
18          System.out.printf("%4d%,20.2f%n", year, amount);
19       }
20    }
21 }
```

```
Year   Amount on deposit
   1            1,050.00
   2            1,102.50
   3            1,157.63
   4            1,215.51
   5            1,276.28
   6            1,340.10
   7            1,407.10
   8            1,477.46
```

```
 9            1,551.33
10            1,628.89
```

Fig. 4.4 | Compound-interest calculations with `for`.

Formatting Strings with Field Widths and Alignment

Line 10 displays headers for two columns of output—the year and the amount on deposit at the end of that year. We use the format specifier `%20s` to output the `String "Amount on Deposit"`. The integer `20` between the `%` and the conversion character `s` indicates that the value should be displayed with a **field width** of 20—that is, `printf` displays the value with at least 20 character positions. If the value to display has fewer than 20 characters (17 in this example), the value is **right aligned** in the field by default. If the value to display had more than 20 characters, the field width would be extended to the right to accommodate the entire value—this would typically upset our tabular output's neat columns. To display values **left aligned**, use the **minus sign (–) formatting flag**, as in `%-20s`.

Performing the Interest Calculations with `static` Method `pow` of Class `Math`

The `for` statement (lines 13–19) executes its body 10 times, varying the control variable `year` from 1 to 10 in increments of 1. This loop terminates when `year` becomes 11. Java does not include an exponentiation operator, but the **`Math` class** has the `static` **`pow` method** for raising a value to a power. Recall from Section 3.12 that you can call a `static` method by specifying the class name followed by a dot (`.`) and the method name. Line 15 implements the formula from the problem statement. The expression

```
Math.pow(1.0 + rate, year)
```

calculates `1.0 + rate` raised to the `year` power. The method receives two `double` arguments and returns a `double` result. Class `Math` is defined in the package `java.lang`, so you do not need to import `Math` to use it. In Chapter 5, you'll learn how to implement custom `static` methods.

The calculation `1.0 + rate` in the `Math.pow` call's first argument produces the same result during each iteration of the loop, so repeating it in every iteration is wasteful. Generally, you should put such calculations outside the loop. However, today's sophisticated

optimizing compilers typically do this when they compile the code.

Formatting Floating-Point Numbers

After each calculation, line 18 outputs the year and the amount on deposit at the end of that year. The `year` is output in a field width of four characters, as specified by `%4d`. The amount is output as a floating-point number with the format specifier `%,20.2f`:[3]

- The **comma (`,`) formatting flag** indicates that the floating-point value should be output with a **grouping separator**. The separator is specific to the user's locale (i.e., country). For example, in the United States, the number will be output using commas to separate every three digits and a decimal point to separate the fractional part of the number, as in 1,050.00.
- The number `20` in the format specification indicates that the value should be right aligned in a field width of 20 characters.
- The precision `.2` tells `printf` to round `amount` to the nearest hundredth and displays it with two digits to the right of the decimal point.

3. Method `printf` uses class `Formatter` to perform its formatting. Class `Formatter`'s documentation provides lists all the formatting capabilities. "Class `Formatter`." Accessed September 21, 2024. https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/util/Formatter.html.

Floating-Point Numbers Are Approximations

Recall from Section 3.7.3 that floating-point numbers often arise in arithmetic as a result of division. For example, when dividing 10 by 3, the result is 3.3333333…, with the sequence of 3s repeating infinitely. The computer allocates a fixed amount of space to hold such a value, so the stored value can be only an approximation. Floating-point types like `double` suffer from what is referred to as **representational error**. Assuming that floating-point numbers are represented exactly can lead to incorrect results.

A Warning about Displaying Rounded Values

In this example, we declared the `double` variables `amount`, `principal` and `rate`. We're dealing with fractional parts of dollars

and thus need a type that allows decimal points in its values. Unfortunately, floating-point numbers can cause trouble. Here's a simple explanation of what can go wrong when using `double` (or `float`) to represent dollar amounts, assuming that dollar amounts are displayed with two digits to the right of the decimal point: Two `double` dollar amounts stored in the machine could be 14.234, which would typically be rounded to 14.23 for display purposes, and 18.673, which would typically be rounded to 18.67 for display purposes. When these amounts are added, they produce the internal sum 32.907, which would typically be rounded to 32.91 for display purposes. Thus, your output could appear as

```
14.23 + 18.67  = 32.91
```

but a person adding the individual numbers as displayed would expect the sum to be 32.90. You've been warned!

Monetary Calculations

Do not use `double` (or `float`) variables for monetary calculations. The imprecision of floating-point numbers can lead to errors. The Java API provides class `BigDecimal` (package `java.math`) for this purpose, which we demonstrate in Section 4.11's Objects-Natural Case Study.

In a global economy, dealing with currencies, monetary amounts, conversions, rounding and formatting is complex. JavaMoney (`https://javamoney.github.io`), which is not part of the core Java API, was developed to meet these challenges.

Generative AI

1 Prompt genAIs with this section's compound-interest problem statement and ask them to write the corresponding Java code, then compare the generated code to the code in Fig. 4.4. As you write your prompt, consider how to specify raising a value to a power. Also, after the genAIs generate their code, provide them with Fig. 4.4's code and ask them to explain any differences between their code and Fig. 4.4.

4.7 `do...while` Iteration Statement

The **`do...while` iteration statement** is similar to the `while` statement:

- The `while` statement tests its loop-continuation condition before executing the loop's body. If the condition is `false`, the body never executes.
- A `do...while` tests its loop-continuation condition after executing the loop's body. Therefore, the body always executes at least once.

Figure 4.5 uses a `do...while` to output the numbers 1–10. Line 6 declares and initializes the control variable `counter`. Upon entering the `do...while` statement, line 9 displays `counter`'s value, and line 10 increments `counter`. Then, the program evaluates the loop-continuation test at line 11. If the condition is `true`, lines 9 and 10 execute again. If the condition is `false`, the loop terminates, and the program continues with the next statement in sequence (line 13).

Click here to view code image

```
1  // Fig. 4.5: DoWhileTest.java
2  // do...while iteration statement.
3
4  public class DoWhileTest {
5     public static void main(String[] args) {
6        int counter = 1;
7
8        do {
9           System.out.printf("%d  ", counter);
10          ++counter;
11       } while (counter <= 10);
12
13       System.out.println();
14    }
15 }
```

```
1 2 3 4 5 6 7 8 9 10
```

Fig. 4.5 | `do...while` iteration statement.

UML Activity Diagram for the `do...while` Iteration Statement

The `do...while` statement's UML activity diagram makes it clear that a `do...while` does not evaluate its loop-continuation condition until after its body executes at least once. Compare this activity diagram with that of the `while` statement shown in Section 3.5.

Generative AI

1 Prompt genAIs, asking them to find and correct the error(s) in the following code segment, which should output the even integers from 2 through 10:

Click here to view code image

```
int counter = 2;

do {
   System.out.println(counter);
   counter += 2;
} While (counter < 10);
```

Confirm that the genAIs recognize the `W` in `while` should be lowercase and the loop-continuation condition's relational operator `<` should be `<=` or the condition should be `counter < 11`.

4.8 `switch` Multiple-Selection Statement

The **`switch` multiple-selection statement** performs different actions based on the possible values of a **constant integral expression** of type `byte`, `short`, `int` or `char` (but not `long`), or based on the value of a `String`. You'll see in later chapters that these restrictions have been eased considerably in recent Java versions.

Using a `switch` Statement to Count A, B, C, D and F Grades

Figure 4.6 calculates the class average of a set of user-entered numeric grades. The program's `switch` statement determines whether each grade is the equivalent of an A, B, C, D or F and increments the appropriate grade counter. Then, the program displays a summary showing the number of students who received

each grade. Like earlier versions of the class-average program, Fig. 4.6 declares local variables `total` (line 7) and `gradeCounter` (line 8) to keep track of the sum of the grades entered by the user and the number of grades entered. Lines 9–13 declare counter variables for each grade category. Note that the variables in lines 7–13 are explicitly initialized to `0`.

Click here to view code image

```
1  // Fig. 4.6: LetterGrades.java
2  // LetterGrades class uses the switch statement to count letter grades.
3  import java.util.Scanner;
4
5  public class LetterGrades {
6     public static void main(String[] args) {
7        int total = 0; // sum of grades
8        int gradeCounter = 0; // number of grades entered
9        int aCount = 0; // count of A grades
10       int bCount = 0; // count of B grades
11       int cCount = 0; // count of C grades
12       int dCount = 0; // count of D grades
13       int fCount = 0; // count of F grades
14
15       Scanner input = new Scanner(System.in);
16
17       System.out.println("""
18          Enter the integer grades in the range 0-100.
19          Enter the end-of-file indicator to terminate input:
20             On macOS/Linux enter <Ctrl> d then press Enter
21             On Windows enter <Ctrl> z then press Enter""");
22
23       // loop until user enters the end-of-file indicator
24       while (input.hasNext()) {
25          int grade = input.nextInt(); // read grade
26          total += grade; // add grade to total
27          ++gradeCounter; // increment number of grades
28
29          //  increment appropriate letter-grade counter
30          switch (grade / 10) {
31             case 10:  // grade was 90-100, inclusive
32             case 9:
33                ++aCount;
34                break; // exits switch
35             case 8: // grade was 80-89
36                ++bCount;
37                break; // exits switch
38             case 7: // grade was 70-79
39                ++cCount;
40                break; // exits switch
```

```
            case 6: // grade was 60-69
               ++dCount;
               break; // exits switch
            default: // grade was less than 60
               ++fCount;
               break; // optional; exits switch anyway
         }
      }

      // display grade report
      System.out.printf("%nGrade Report:%n");

      // if user entered at least one grade...
      if (gradeCounter != 0) {
         // calculate average of all grades entered
         double average = (double) total / gradeCounter;

         // output summary of results
         System.out.printf("Total of the %d grades entered is %d%n",
            gradeCounter, total);
         System.out.printf("Class average is %.2f%n", average);
         System.out.printf("%n%s%n%s%d%n%s%d%n%s%d%n%s%d%n%s%d%n",
            "Number of students who received each grade:",
            "A: ", aCount,  // display number of A grades
            "B: ", bCount,  // display number of B grades
            "C: ", cCount,  // display number of C grades
            "D: ", dCount,  // display number of D grades
            "F: ", fCount); // display number of F grades
      }
      else { // no grades were entered, so output appropriate message
         System.out.println("No grades were entered");
      }
   }
}
```

```
Enter the integer grades in the range 0-100.
Enter the end-of-file indicator to terminate input:
   On macOS/Linux enter <Ctrl> d then press Enter
   On Windows enter <Ctrl> z then press Enter
99
92
45
57
63
71
76
85
90
100
^z
```

```
Grade Report:
Total of the 10 grades entered is 778
Class average is 77.80

Number of students who received each grade:
A: 4
B: 1
C: 2
D: 1
F: 2
```

Fig. 4.6 | `LetterGrades` class uses the `switch` statement to count letter grades.

Method `main` has two key parts. Lines 24–48 read an arbitrary number of integer grades from the user using sentinel-controlled iteration, update instance variables `total` and `gradeCounter`, and increment an appropriate letter-grade counter for each grade entered. Lines 51–69 output a report containing the total of all grades entered, the average of the grades and the number of students who received each letter grade. If no grades were entered, line 71 displays an appropriate message. Let's examine these parts in more detail.

Reading Grades from the User

Lines 17–21 prompt the user to enter integer grades and to enter the end-of-file indicator to terminate the input. Here, we specify the instructions using a relatively new Java feature called a **text block**—a convenient way to represent a **multiline string**. Each text block begins with the delimiter `"""` on a line by itself, followed by any number of lines of text and a terminating `"""` delimiter. Each line of text (except the one with the terminating `"""`) ends with an implicit newline.

The **end-of-file indicator** is a system-dependent keystroke combination the user enters to indicate there's no more data to input. On macOS and Linux systems, you can enter end-of-file by inputting the following sequence on a line by itself:

<Ctrl> d

This notation means to simultaneously press both the *Ctrl* key and the *d* key. On Windows systems, you can enter end-of-file by inputting

```
<Ctrl> z
```

On some systems, you must press *Enter* after inputting the end-of-file key sequence. Also, Windows typically displays the characters `^Z` on the screen when the end-of-file indicator is entered, as shown in Fig. 4.6's sample output.

The `while` statement (lines 24–48) obtains the user input. The condition at line 24 calls `Scanner`'s **`hasNext` method** to determine whether there's more data to input. This method returns `true` if there's more data in the input stream; otherwise, it returns `false`. The returned value is then used as the `while` statement's condition. When using `hasNext` with the standard-input stream, line 24 waits for the user to enter a value or the end-of-file indicator, then the program continues accordingly. The `hasNext` method returns `false` once the user enters the end-of-file indicator.

Line 25 inputs a grade from the user. Line 26 adds `grade` to `total`. Line 27 increments `gradeCounter`. These variables are used to compute the average of the grades.

Processing the Grades

The `switch` statement (lines 30–47) determines which letter-grade counter to increment based on the numeric grade entered. We assume that the user enters a valid grade in the range 0–100 where 90–100 represents A, 80–89 represents B, 70–79 represents C, 60–69 represents D, and 0–59 represents F. The `switch` statement's block contains a sequence of **`case` labels** and an optional **`default` case**, which can appear anywhere in the `switch` but typically appears last. When the flow of control reaches the `switch`, the program evaluates the **controlling expression** in the parentheses following `switch`

```
grade / 10
```

then compares this value with each `case` label.

Controlling Expression Produces Integer Values

Line 30's controlling expression performs integer division, which truncates the result's fractional part. When we divide an `int` value from 0 to 100 by 10, the result is always an `int` value from 0 to 10. We test several of these values in our `case` labels. If the user enters the integer `85`, the controlling expression evaluates to 8. The `switch` compares `8` with each `case` label. If a match occurs (`case`

`8:` at line 35), that `case`'s statements execute. For `8`, line 36 increments `bCount`, because a grade in the 80s is a B. The **`break` statement** (line 37) exits the `switch`, then the next statement in sequence executes. In this program, we reach the end of the `while` loop, so control returns to its loop-continuation condition in line 24 to determine whether the loop should continue executing.

Performing the Same Statements for Multiple `case`s

Our `switch` statement's `case`s explicitly test for the values `10`, `9`, `8`, `7` and `6`. The `case`s in lines 31–32 test for the values `10` and `9`, which both represent the grade A. Listing `case`s consecutively with no statements between them enables them to perform the same set of statements. So, when the controlling expression evaluates to `10` or `9`, the statements in lines 33–34 execute. The `switch` statement does not provide a mechanism for testing ranges of values—every value you need to test must be listed in a separate `case` label.[4] The `switch` statement differs from other control statements in that it does not require braces around multiple statements in a `case`.

4. In Chapter 5, we introduce `switch` expressions, which can specify in each `case` a comma-delimited list of values to match.

The `default` Case

If the controlling expression's value does not match any of the `case` labels, the `default` case (lines 44–46) executes. In this example, the `default` case processes all controlling-expression values less than `6`—that is, all failing grades. If no match occurs and the `switch` does not contain a `default` case, program control simply continues with the first statement after the `switch`. In a `switch`, it's good practice to test all possible values of the controlling expression. Also, providing a `default` case focuses you on the need to process exceptional conditions.

Displaying the Grade Report

Lines 51–69 output a report based on the grades entered (as shown in the input/output window in Fig. 4.6). Line 54 determines whether the user entered at least one grade—this helps us avoid dividing by zero. If so, line 56 calculates the class average. Lines 59–68 then output the total of all the grades, the class average and the number of students who received each letter grade. If no grades were entered, line 71 outputs an appropriate message.

`case` without a `break` Statement

Without `break` statements, each time a match occurs in the `switch`, the statements for that `case` and subsequent `case`s execute until a `break` statement or the end of the `switch` is reached. This is referred to as "falling through" to the statements in subsequent `case`s. Forgetting a `break` statement when one is needed in a `switch` is a logic error.

`switch` Statement UML Activity Diagram

The following is the UML activity diagram for the general `switch` statement with `break`s:
Most `switch` statements use a `break` in each `case` to terminate the `switch` statement. The diagram emphasizes this by including `break` statements and showing that the `break` statement at the end of a `case` causes control to exit the `switch` statement immediately. A `break` is not required for the `switch`'s last `case` (or the optional `default` case when it appears last)—execution simply continues with the next statement after the `switch`.

Notes on the Expression in Each `case` of a `switch`

In a `switch`, each `case` must contain a `String` or a constant integral expression—that is, any combination of integer constants that evaluates to a constant integer value (e.g., –7, 0 or 221) of type `byte`, `char`, `short` or `int`. An integer constant is simply an integer value. In addition, you can use **character constants** (of type `char`), which are specific characters in single quotes, such as `'A'`, `'7'` or `'$'`. These represent the characters' integer values in the Unicode character set.[5] You can view the integer values of many Unicode characters at

Click here to view code image

```
https://en.wikipedia.org/wiki/List_of_Unicode_characters
```

5. "Unicode." Accessed May 26, 2024. https://home.unicode.org/.

The expression in each `case` can also be a constant variable or an `enum` constant:

- A **constant variable** (or simply a **constant**) is declared `final` (discussed in Chapter 5) and contains a value that does not change throughout the program.
- Java also has a feature called `enum` types (discussed in

Chapter 5) that define constants. An `enum`'s constants may be used in `case` labels.

In Chapter 9, Real-World Modeling with Inheritance, Polymorphism & Interfaces, we present a more elegant way to implement `switch` logic. We use a technique called polymorphism to create programs that often are clearer, easier to maintain and easier to extend than programs using `switch` logic.

Generative AI

1 Prompt genAIs, asking them to find and correct the error(s) in the following code:

Click here to view code image

```
switch (number) {
   case 1:
      System.out.println("The number is 1\n");
   case 2:
      System.out.println("The number is 2\n");
      break;
   default:
      System.out.println("The number is not 1 or 2\n");
      break;
}
```

Confirm that the genAIs recognize the missing `break` statement in the first `case`—if the value of `number` is `1`, the output statements in `case`s `1` and `2` will both execute. To fix this, the genAI should add a `break` statement after the `println` statement in the first `case`.

4.9 `break` and `continue` Statements

In addition to selection and iteration statements, Java provides the `break` and `continue` statements to alter the flow of control.[6]

6. Java also has labeled `break` and `continue` statements. See the book's webpage at `https://deitel.com/javafp5`.

`break` Statement

In the preceding section, we used `break` to terminate a `switch`

statement's execution. Executing a `break` in a `while`, `for` or `do...while` immediately exits that statement—execution continues with the next statement in sequence. Figure 4.7 demonstrates `break` exiting a `for`. When the nested `if` statement (lines 8–10) detects that `count` is `5`, the `break` statement (line 9) terminates the enclosing loop, then the program proceeds to line 15, which displays a message indicating the control variable's value when the loop terminated. The loop fully executes its body only four times instead of the 10 specified in the `for` header (line 7). Line 15 can access the control variable `count` after the loop because we declared `count` before the loop (line 5) rather than in the `for` header's initialization section (line 7).

Click here to view code image

```
1  // Fig. 4.7: BreakTest.java
2  // break statement exiting a for statement.
3  public class BreakTest {
4     public static void main(String[] args) {
5        int count; // control variable also used after loop terminates
6
7        for (count = 1; count <= 10; ++count) { // loop 10 times
8           if (count == 5) {
9              break; // terminates loop if count is 5
10          }
11
12          System.out.printf("%d ", count);
13       }
14
15       System.out.printf("%nBroke out of loop at count = %d%n", count);
16    }
17 }
```

```
1 2 3 4
Broke out of loop at count = 5
```

Fig. 4.7 | `break` statement exiting a `for` statement.

`continue` Statement

When you execute the `continue` statement in a loop, it skips the remaining statements in the loop's body and proceeds with the loop's next iteration:

- In a `while` or `do...while` statement, the program evaluates the loop-continuation test immediately after `continue`

executes.

- In a `for` statement, the increment expression executes, then the program evaluates the loop-continuation test.

Figure 4.8 uses `continue` (line 7) to skip the statement at line 10 when the nested `if` determines that `count`'s value is `5`. When the `continue` executes, program control continues with the `++count` increment expression in the `for` statement's header (line 5).

Click here to view code image

```
1  // Fig. 4.8: ContinueTest.java
2  // continue statement terminating an iteration of a for statement.
3  public class ContinueTest {
4     public static void main(String[] args) {
5        for (int count = 1; count <= 10; ++count) { // loop 10 times
6           if (count == 5) {
7              continue; // skip remaining code in loop body if count is 5
8           }
9
10          System.out.printf("%d ", count);
11       }
12
13       System.out.printf("%nUsed continue to skip printing 5%n");
14    }
15 }
```

```
1 2 3 4 6 7 8 9 10
Used continue to skip printing 5
```

Fig. 4.8 | `continue` statement terminating an iteration of a `for` statement.

Some programmers feel that the `break` and `continue` statements violate structured programming. The same effects are achievable with structured-programming techniques, so these programmers do not use `break` or `continue`.

There's a tension between achieving quality software engineering and achieving the best-performing software. Sometimes, one goal is achieved at the expense of the other. For all but the most performance-intensive situations, apply the following guidelines: First, make your code simple and correct; then make it fast and small, but only if necessary.

Generative AI

1 Prompt genAIs, asking them to reimplement Fig. 4.8's code using a structured approach, then compare the code they write to the following code:

Click here to view code image

```
for (int count = 1; count <= 10; ++count) { // loop 10 times
   if (count != 5) {
      System.out.printf("%d ", count);
   }
}
```

4.10 Logical Operators

The `if`, `if...else`, `while`, `do...while` and `for` statements each require a condition to determine how to continue a program's flow of control. So far, we've studied only simple conditions, such as `count <= 10` and `total > 1000`. Simple conditions test only one condition and are expressed in terms of the relational operators `>`, `<`, `>=` and `<=` and the equality operators `==` and `!=`. To test multiple conditions when making a decision, we performed these tests in separate statements or in nested `if` or `if...else` statements. Sometimes, control statements require more complex conditions to determine a program's flow of control.

The **logical operators**

- `&&` (conditional AND),
- `||` (conditional OR),
- `&` (boolean logical AND),
- `|` (boolean logical inclusive OR),
- `^` (boolean logical exclusive OR) and
- `!` (logical complement)

enable you to combine two or more simple conditions to form **compound conditions**. The `&`, `|` and `^` operators are also bitwise operators when they're applied to integer operands.7

7. See the book's webpage at `https://deitel.com/javafp5` for a discussion of bitwise operators.

4.10.1 Conditional AND (`&&`) Operator

In 2023, after three years of the Covid pandemic, the United States experienced a pilot shortage that snarled air traffic. To alleviate the pilot shortage, the government considered raising the mandatory pilot retirement age from 65 to 67 so veteran pilots in good health could continue flying for two more years.

Let's assume an airline wanted to ask each veteran pilot aged 63 or older whether they'd like to apply to keep flying until age 67 rather than retiring at age 65, and the airline needed to know how many pilots fit into this age category. The airline could write a program to search their employee records and determine the number of 63-years-of-age-or-older veteran pilots. The program might contain the statement:

Click here to view code image

```
if (employeeType == PILOT && age >= 63) {
   ++veteranPilots;
}
```

Assume `PILOT` is a constant. The `if` statement contains two simple conditions separated by the **`&&` (conditional AND) operator**:

- `employeeType == PILOT` determines whether an employee is a pilot.
- `age >= 63` determines whether an employee is nearing retirement age.

The `if` statement considers the combined condition, which is `true` only if both simple conditions are `true`. If so, the `if` statement's body increments `veteranPilots` by `1`; otherwise, the program skips the increment. Some programmers find that the combined condition is more readable with redundant parentheses, as in

Click here to view code image

```
if ((employeeType == PILOT) && (age >= 63)) {
   ++veteranPilots;
}
```

The following **truth table** summarizes the `&&` operator, showing all possible combinations of `false` and `true` values for *expression1* and *expression2*. Java evaluates to `false` or `true` all expressions that include relational operators, equality operators or logical operators:

Conditional AND (&&) Operator Truth Table

expression1	&&	*expression2*
false	&&	false
false	&&	true
true	&&	false
true	&&	true

4.10.2 Conditional OR (||) Operator

Suppose we wish to ensure that either or both of two conditions are `true` before choosing a path of execution. In this case, we use the **|| (conditional OR) operator**, as in the following program segment:

Click here to view code image

```
if ((semesterAverage >= 90) || (finalExam >= 90)) {
   System.out.println ("Student grade is A");
}
```

This statement also contains two simple conditions:

- `semesterAverage >= 90` determines whether the student deserves an A in the course for a solid performance throughout the semester.
- `finalExam >= 90` determines whether the student deserves an A in the course for outstanding performance on the final exam.

The `if` statement then considers the combined condition

Click here to view code image

```
(semesterAverage >= 90) || (finalExam >= 90)
```

and awards the student an A if either or both of the simple conditions are `true`. The only time this would not print `"Student grade is A"` is if both simple conditions are `false`. The following is a truth table for the conditional OR (`||`) operator.

Conditional OR (||) Operator Truth Table

expression1	||	*expression2*
false	||	false
false	||	true
true	||	false
true	||	true

The `&&` operator has a higher precedence than `||`. Both group from left to right.

4.10.3 Short-Circuit Evaluation of Complex Conditions

The parts of an expression containing `&&` or `||` operators are evaluated only until it's known whether the condition is `true` or `false`. Thus, the evaluation of the condition

Click here to view code image

```
(employeeType == PILOT) && (age >= 63)
```

stops immediately if `employeeType` is not equal to `PILOT` (i.e., the entire expression is `false`) but continues if `employeeType` is equal to `PILOT` (i.e., the entire expression might still be `true` if `age >= 63` is `true`). This **short-circuit evaluation** is a performance feature of `&&` and `||` expressions.

In expressions using `&&`, a condition—we'll call this the **dependent condition**—may require another condition to be `true` for the evaluation of the dependent condition to be meaningful. In this case, the dependent condition should be placed after the `&&` operator to prevent errors. Consider the following condition:

```
(i != 0) && (10 / i == 2)
```

The dependent condition `(10/i==2)` must appear after the `&&` operator to prevent the possibility of division by zero.

4.10.4 Boolean Logical AND (&) and Inclusive OR (|) Operators

The **boolean logical AND (&) operator** and **boolean logical inclusive OR (|) operator** are identical to the `&&` and `||` operators, except that the `&` and `|` operators always evaluate both operands—they do not perform short-circuit evaluation. This is useful if the right operand has a required **side effect**—a modification of a variable's value. For example,

Click here to view code image

```
(birthday == true) | (++age >= 65)
```

guarantees that the condition `++age >= 65` will be evaluated. Thus, the variable `age` is incremented, regardless of whether the overall condition is `true` or `false`. For clarity, avoid expressions with side effects (such as increments) in conditions. They can make code harder to understand and can lead to subtle logic errors.

Assignment (=) expressions generally should not be used in conditions. Every condition must result in a `boolean` value; otherwise, a compilation error occurs. In a condition, an assignment will compile only if a `boolean` expression is assigned to a `boolean` variable.

4.10.5 Boolean Logical Exclusive OR (^) Operator

A simple condition containing the **boolean logical exclusive OR (^) operator** is `true` only if one of its operands is `true` and the other is `false`. If both are `true` or both are `false`, the condition is `false`. This operator is guaranteed to evaluate both of its operands. The following is a truth table for the boolean logical exclusive OR (^) operator.

Boolean Logical Exclusive OR (^) Operator Truth Table

expression1	^	*expression2*
false	^	false
false	^	true
true	^	false
true	^	true

4.10.6 Logical Complement (!) Operator

The **logical complement (!) operator**—also called the **logical negation operator** or **logical NOT operator**—"reverses" the meaning of a condition. Unlike the logical operators `&&`, `||`, `&`, `|` and `^`, which are binary operators that combine two conditions, the logical negation operator is a unary operator with only one condition as an operand. The operator is placed before a condition to choose a path of execution if the original condition (without the logical negation operator) is `false`, as in the program segment

Click here to view code image

```
if (!(grade == sentinelValue)) {
   System.out.printf("The next grade is %d%n", grade);
```

```
}
```

which executes the `printf` call only if `grade` is not equal to `sentinelValue`. The parentheses around the condition `grade == sentinelValue` are needed because the logical negation operator has a higher precedence than the equality operator.

In most cases, you can avoid using logical negation by expressing the condition differently with an appropriate relational or equality operator. For example, the previous statement also may be written as follows:

Click here to view code image

```
if (grade != sentinelValue) {
   System.out.printf("The next grade is %d%n", grade);
}
```

This flexibility can help you express a condition more conveniently. The following is a truth table for the logical negation operator.

Logical Complement (`!`) Operator Truth Table

expression	*!expression*
false	true
true	false

4.10.7 Example: Generating Logical-Operator Truth Tables

Figure 4.9 uses logical operators to produce the truth tables discussed in this section. The output shows each `boolean` expression and its result. We used the `%b` (`boolean`) format specifier to display the word "true" or the word "false" based on a `boolean` expression's value. Lines 7–11 produce the truth table for `&&`. Lines 14–18 produce the truth table for `||`. Lines 21–25 produce the truth table for `&`. Lines 28–33 produce the truth table for `|`. Lines 36–41 produce the truth table for `^`. Lines 44–45 produce the truth table for `!`.

Click here to view code image

```
1  // Fig. 4.9: LogicalOperators.java
2  // Logical operators.
3
4  public class LogicalOperators {
```

```
5       public static void main(String[] args) {
6          // create truth table for && (conditional AND) operator
7          System.out.printf("%s%n%s: %b%n%s: %b%n%s: %b%n%n",
8             "Conditional AND (&&)", "false && false", (false && false),
9             "false && true", (false && true),
10            "true && false", (true && false),
11            "true && true", (true && true));
12
13         // create truth table for || (conditional OR) operator
14         System.out.printf("%s%n%s: %b%n%s: %b%n%s: %b%n%n",
15            "Conditional OR (||)", "false || false", (false || false),
16            "false || true", (false || true),
17            "true || false", (true || false),
18            "true || true", (true || true));
19
20         // create truth table for & (boolean logical AND) operator
21         System.out.printf("%s%n%s: %b%n%s: %b%n%s: %b%n%n",
22            "Boolean logical AND (&)", "false & false", (false & false),
23            "false & true", (false & true),
24            "true & false", (true & false),
25            "true & true", (true & true));
26
27         // create truth table for | (boolean logical inclusive OR) operator
28         System.out.printf("%s%n%s: %b%n%s: %b%n%s: %b%n%n",
29            "Boolean logical inclusive OR (|)",
30            "false | false", (false | false),
31            "false | true", (false | true),
32            "true | false", (true | false),
33            "true | true", (true | true));
34
35         // create truth table for ^ (boolean logical exclusive OR) operator
36         System.out.printf("%s%n%s: %b%n%s: %b%n%s: %b%n%n",
37            "Boolean logical exclusive OR (^)",
38            "false ^ false", (false ^ false),
39            "false ^ true", (false ^ true),
40            "true ^ false", (true ^ false),
41            "true ^ true", (true ^ true));
42
43         // create truth table for ! (logical negation) operator
44         System.out.printf("%s%n%s: %b%n%s: %b%n", "Logical NOT (!)",
45            "!false", (!false), "!true", (!true));
46      }
47   }
```

```
Conditional AND (&&)
false && false: false
false && true: false
true && false: false
true && true: true
```

```
Conditional OR (||)
false || false: false
false || true: true
true || false: true
true || true: true

Boolean logical AND (&)
false & false: false
false & true: false
true & false: false
true & true: true

Boolean logical inclusive OR (|)
false | false: false
false | true: true
true | false: true
true | true: true

Boolean logical exclusive OR (^)
false ^ false: false
false ^ true: true
true ^ false: true
true ^ true: false

Logical NOT (!)
!false: true
!true: false
```

Fig. 4.9 | Logical operators.

Generative AI

1 Prompt genAIs, asking them to write a program that displays the `boolean` results of the following conditions assuming `i = 1`, `j = 2`, `k = 3` and `m = 2`:

a. `i == 1`
b. `j == 3`
c. `(i >= 1) && (j < 4)`
d. `(m <= 99) & (k < m)`
e. `(j >= i) || (k == m)`
f. `(k + m < j) | (3 - j >= k)`
g. `!(k > m)`

Run their generated code to confirm the conditions produce the following results: a) `true`. b) `false`. c) `true`. d) `false`. e) `true`. f)

`false`. g) `false`.

4.11 Objects-Natural Case Study: Precise Monetary Calculations with Java API Class `BigDecimal`

Many applications require precise representations of numbers with decimal points. Institutions like banks that deal with millions or even billions of transactions daily must tie out their monetary transactions "to the penny." We mentioned earlier that using `double` values to process monetary amounts, as in Fig. 4.4's compound-interest example, introduces representational error and rounding issues that could prevent "to the penny" accuracy:

- There were representational errors because we stored precise decimal monetary amounts and interest rates as `double`s, which are only approximations.
- The floating-point values were rounded because the interest calculations often result in fractional pennies, which must be rounded to the hundredths place to represent cents. Rounding can be unpredictable in situations with floating-point representational errors.

There are many points in monetary calculations in which rounding may occur. For example, on a restaurant bill, the tax could be calculated on each item, resulting in separate rounding operations, or it could be calculated only once on the total bill amount. These alternate approaches could yield different results.

Custom Types for Precise Monetary Calculations

You'll see in Chapter 8 that Java is an extensible language. So, we could create a custom class for monetary calculations and other applications that require precise representation and manipulation of numbers with decimal points. However, getting it right might take significant time and effort. Rather than "reinventing the wheel," we can use preexisting classes from the Java API (or other libraries) that provide types for to-the-penny precision. In this Objects-Natural Case Study, we'll create objects of the **Java API's**

`BigDecimal` **class** and use its methods to perform precise monetary calculations.

Representing and Calculating with Monetary Amounts Is a Complex Issue

Processing monetary amounts can be far more complex than what we show here. Banks have to deal with issues such as using a "fair rounding algorithm" when calculating daily interest on their accounts. Rounding rules can vary by currency, government, company, etc. Also, there are a great many currencies worldwide with different conventions for thousands separators, decimal separators, currency symbols, and more. Various Java libraries (including ones that are not part of the core Java API, such as JavaMoney) deal with issues like these.

`BigDecimal` Objects Require More Memory than Primitive Type Variables

`BigDecimal` can represent exact monetary amounts like 123.02 and perform to-the-penny arithmetic without representational error. To do this, `BigDecimal`'s internal coding scheme uses extra memory to store values with arbitrary precision. A `BigDecimal`'s precision includes all the significant digits to the left and right of the decimal point. By default, `BigDecimal` calculations are exact—they do not round their results. As you'll see, you can specify a **rounding policy**, which financial institutions typically require for monetary calculations.

The following statement creates the `BigDecimal` variable `monetaryAmount` and initializes it with the `double` value `123.02` that we mentioned above:

Click here to view code image

```
BigDecimal monetaryAmount = new BigDecimal(123.02);
```

If you print `monetaryAmount`, you'll see that this `BigDecimal` number is actually initialized from the following `double` value "under the hood:"

Click here to view code image

```
123.01999999999999602096067974343895912170410156 25
```

This clearly shows this `double`'s representational error—

`monetaryAmount` is initialized to slightly less than 123.02. To represent 123.02 precisely, initialize `monetaryAmount` with a `String`, as in:

Click here to view code image

```
BigDecimal monetaryAmount = new BigDecimal("123.02");
```

Demonstrating the Java API's `BigDecimal` Class

Figure 4.10 reimplements the compound interest calculation in Fig. 4.4 using `BigDecimal` objects to represent monetary amounts and interest rates precisely and perform precise arithmetic using them. Lines 3–4 `import` the **`java.math` package** classes we'll use in this example. Like Fig. 4.4, the first two output columns in Fig. 4.10 display the year and the rounded amount on deposit at the end of each year. This example also displays a third column showing the exact stored value. You can use this for comparison purposes when we discuss how rounding works. Specifying a rounding policy causes a slight difference in the second column's displayed values vs. those displayed in Fig. 4.4.

Click here to view code image

```
1  // Fig. 4.11: BigDecimalInterest.java
2  // Compound-interest example with BigDecimal objects.
3  import java.math.BigDecimal;
4  import java.math.RoundingMode;
5
6  public class BigDecimalInterest {
7     public static void main(String[] args) {
8        // create BigDecimal objects from strings
9        BigDecimal principal = new BigDecimal("1000.00");
10       BigDecimal rate = new BigDecimal("0.05");
11
12       System.out.printf("Initial principal: %7s%n", principal);
13       System.out.printf("    Interest rate: %7s%n%n", rate);
14
15       // display headers
16       System.out.printf("%s%17s   %s%n",
17          "Year", "Rounded Amount", "Precise Stored Amount");
18
19       // calculate amount on deposit for each of 10 years
20       for (int year = 1; year <= 10; ++year) {
21          BigDecimal amount = principal.multiply(
22             BigDecimal.ONE.add(rate).pow(year));
23          System.out.printf("%4d%17s   %s%n", year,
24             amount.setScale(2, RoundingMode.HALF_EVEN), amount);
```

```
25          }
26       }
27    }
```

```
Initial principal: 1000.00
    Interest rate:    0.05

Year    Rounded Amount    Precise Stored Amount
   1           1050.00    1050.0000
   2           1102.50    1102.500000
   3           1157.62    1157.62500000
   4           1215.51    1215.5062500000
   5           1276.28    1276.281562500000
   6           1340.10    1340.09564062500000
   7           1407.10    1407.1004226562500000
   8           1477.46    1477.455443789062500000
   9           1551.33    1551.32821597851562500000
  10           1628.89    1628.894626777414062500000
```

Fig. 4.10 | Compound-interest example with `BigDecimal` objects.

Initializing `BigDecimal` Objects

Due to the representational error with floating-point numbers, you typically create a `Big-Decimal` object from

- an integer value, which is represented exactly by Java, or
- a string containing the exact digits you wish to represent to the left and right of the decimal point.

Lines 9 and 10 initialize the `BigDecimal`s `principal` and `rate` with the strings `"1000.00"` and `"0.05"`, respectively.

Formatting `BigDecimal` Objects for Output

`BigDecimal`s are converted to strings for display purposes. By default, the digits of their exact values are displayed. Lines 12–13 use the `%7s` format specifier to right-align the `principal` and `rate` in fields that are seven characters wide. The `principal` and `rate` decimal points align vertically because each has two digits to the right of the decimal point.

Performing the Interest Calculations with `BigDecimal`

Lines 21–22

Click here to view code image

```
BigDecimal amount = principal.multiply(
   BigDecimal.ONE.add(rate).pow(year));
```

perform the interest calculation using `BigDecimal`'s **`multiply`**, **`add`** and **`pow`** **methods**. The expression evaluates as follows:

- First, the expression `BigDecimal.ONE.add(rate)` adds `1` to the `rate` to produce a `BigDecimal` containing `1.05`—equivalent to `1.0 + rate` in line 15 of Fig. 4.4. **`BigDecimal.ONE`** is a predefined `BigDecimal` object representing the value `1`. `Big-Decimal` also predefines **`BigDecimal.ZERO`** (`0`), **`BigDecimal.TWO`** (`2`) and **`BigDecimal.TEN`** (`10`). These values are used frequently in `BigDecimal` calculations and help programmers avoid the overhead of creating them repeatedly.
- Next, we call `BigDecimal` method `pow`—which receives an `int` argument—on the preceding result to raise `1.05` to the power `year`. This is equivalent to passing `1.0 + rate` and `year` to method `Math.pow` in line 15 of Fig. 4.4.
- Finally, we call `BigDecimal` method `multiply` on the `principal` object, passing the preceding result as the argument. This returns a `BigDecimal` representing the amount on deposit at the end of the specified `year`.

Lines 23–24 display the year, the rounded `amount` on deposit at the end of each year and, for comparison purposes, the unrounded `amount`, which in year 10 already contains much more precision than `double` offers. We discuss the rounding below.

A Policy Issue: Half-Up Rounding vs. Banker's Rounding

Compare Fig. 4.4's and Fig. 4.10's outputs. Year `3` has a slightly different output—`1157.63` in Fig. 4.4 vs. `1157.62` in Fig. 4.10. This is due to the different rounding algorithms used for `double`s vs. the `BigDecimal`s in this example and could also be due to representational errors when calculating with `double`s. In Fig. 4.4's calculation for year `3`,

```
1000.00 * pow(1 + 0.05, 3)
```

`pow` returns 1.157625 (that is, 1.05^3), so the preceding calculation

yields 1157.625. By default, `double` amounts use **half-up rounding**—for two digits to the right of the decimal point, if the third digit is 5 or higher, we round the previous digit up; otherwise, we round it down. So in Fig. 4.4, 1157.625 rounded up to 1157.63.

Banker's Rounding

Unfortunately, the commonly used half-up rounding is a biased technique:

- For the four digits 1, 2, 3 and 4, we round the preceding digit down.
- For the five digits 5, 6, 7, 8 and 9, we round the preceding digit up.

More values round up than down, leading to possible discrepancies in monetary calculations. To fix this bias, the expression

Click here to view code image

```
amount.setScale(2, RoundingMode.HALF_EVEN)
```

in line 24 uses `BigDecimal`'s **`setScale` method** to round `amount` to two digits to the right of the decimal point using **banker's rounding**[8] (also called **round half to even** or simply **half even**)—specified by `setScale`'s second argument, **`RoundingMode.HALF_EVEN`**.

8. "Rounding," Wikipedia. Wikimedia Foundation. Accessed June 7, 2024. https://en.wikipedia.org/wiki/Rounding.

With banker's rounding, when the digit used to determine rounding is 5 (the third digit in this example), this algorithm rounds the previous digit (the second digit in this example) to the nearest even integer. So, when rounding the `BigDecimal amount`, the value

- 1157.625 **rounds down** to 1157.62. Similarly,
- 1157.635 would **round up** to 1157.64,
- 1157.645 would **round down** to 1157.64,
- 1157.655 would **round up** to 1157.66,
- 1157.665 would **round down** to 1157.66,
- etc.

Generative AI

1 Prompt genAIs, asking them to list the most popular Java libraries for handling monetary amounts. Ask them to tell you how many stars each of the Java libraries they recommended has on GitHub—larger numbers of stars often indicate a library's popularity in the open-source software development community.

4.12 Wrap-Up

We've now introduced most of Java's control statements, which enable you to control the flow of execution in functions. Chapter 3 discussed `if`, `if...else` and `while`. This chapter demonstrated `for`, `do...while` and `switch`. In Chapter 6, we'll introduce the enhanced `for` statement.

You used the `break` statement to exit a `switch` statement and to terminate a loop immediately. You used the `continue` statement to terminate a loop's current iteration and proceed with the loop's next iteration. We mentioned that `while` and `do...while` loops proceed immediately to their loop-continuation conditions, whereas `for` loops proceed to their increments first. We introduced the logical operators for creating more complex conditional expressions in control statements. You also continued using generative AI to leverage your Java learning experience.

In our first compound-interest example, we discussed the representational errors associated with floating-point types. Then, in the Objects-Natural Case Study, we performed precise monetary calculations using the Java API's `BigDecimal` class to represent floating-point numbers exactly and perform "to-the-penny" arithmetic. We showed that primitive type `double` uses the biased half-up rounding algorithm and that `BigDecimal` eliminates that bias by using banker's rounding.

In Chapter 5, you'll create your own custom methods, understand how the method-call-and-return mechanism works, learn simulation techniques with random-number generation, and define overloaded methods—that is, methods with the same name that perform the same jobs but with different types and/or numbers of arguments.

Special Section: Generative AI Exercises

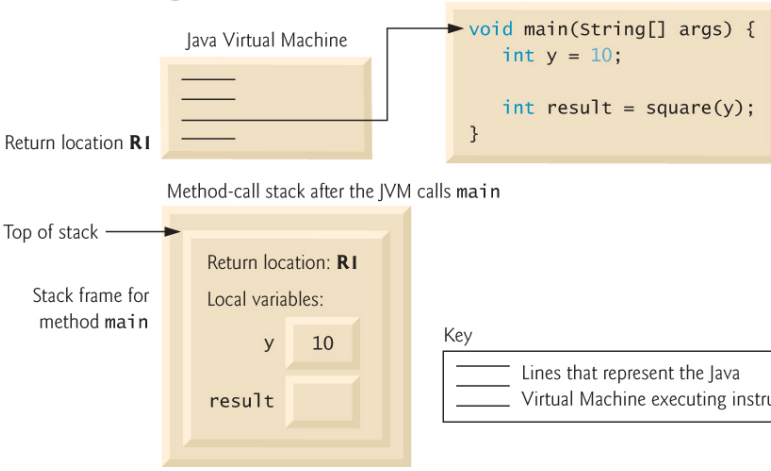
For each of the following questions, try multiple genAIs to determine which performs best. Always verify the results. Keep in

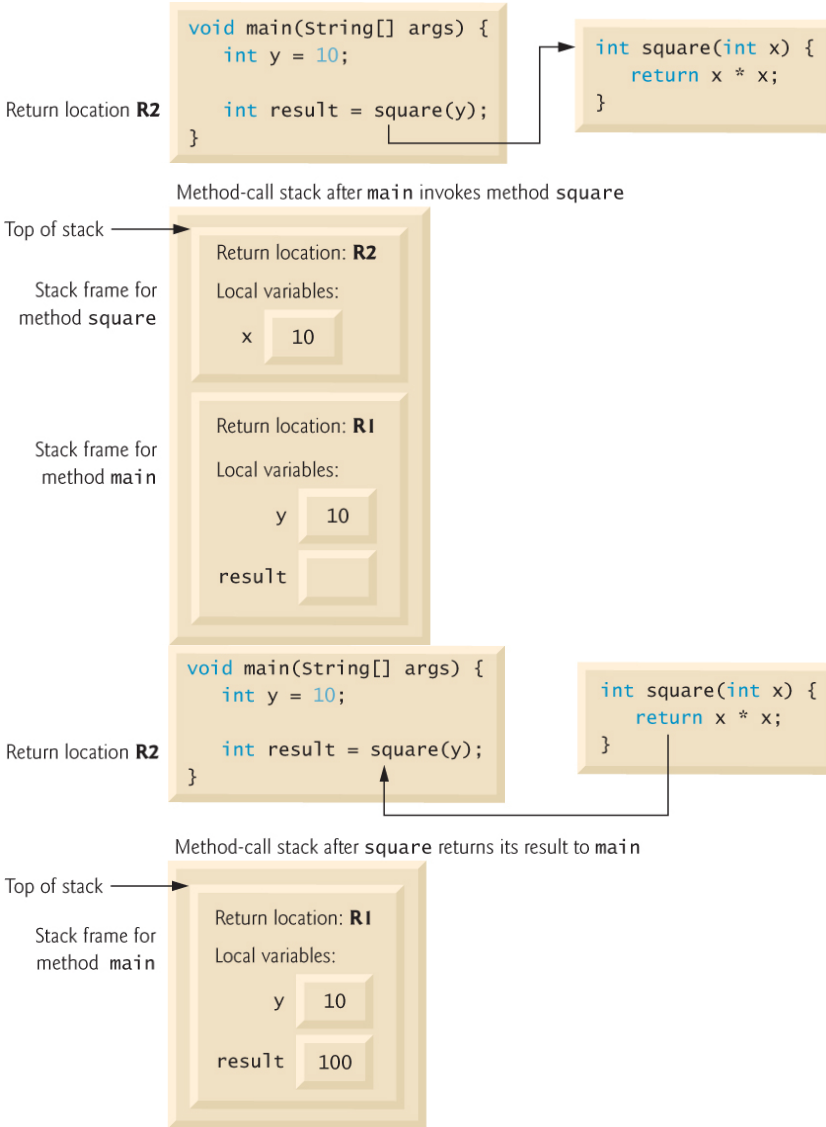
mind that they sometimes make mistakes and even hallucinate, creating unreliable information.

4.1 ***(Generative AI: Control Statements Exercises)*** Prompt generative AIs to suggest introductory exercises with answers for practice with Java's `for`, `do`...`while` and `switch` control statements. Limit the suggestions to exercises that can be completed in a maximum of 50 lines of code.

4.2 ***(Generative AI: Challenges of Processing Monetary Amounts)*** Prompt generative AIs to explain why processing monetary amounts in programming languages is challenging.

4.3 ***Generative AI: Processing Monetary Amounts Precisely in Java)*** Prompt generative AIs to list the various ways to process monetary amounts precisely in Java.


Java Virtual Machine
Return location RI
void main(String[] args) {
int y = 10;
int result = square(y);
}
Method-call stack after the JVM calls main
Top of stack
Return location: RI
Local variables:
y
10
result
Stack frame for method main
Key
Lines that represent the Java Virtual Machine executing instructions


void main(String[] args) {
int y = 10;
int result = square(y);
}
int square(int x) {
return x * x;
}
Return location R2
Method-call stack after main invokes method square
Top of stack
Return location: R2
Local variables:
x
10
Stack frame for method square
Return location: R1
Local variables:
y
10
result
Stack frame for method main
void main(String[] args) {
int y = 10;
int result = square(y);
}
int square(int x) {
return x * x;
}
Return location R2
Method-call stack after square returns its result to main
Top of stack
Return location: R1
Local variables:
y
10
result
100
Stack frame for method main

Methods

Objectives

In this chapter, you'll:

- Create custom methods.
- Use random-number generation for simulation and game-playing programs.
- Use `switch` statements with new, more concise `case` labels.
- Use `switch` expressions to assign values to variables.
- Learn that the visibility of declarations is limited to specific regions of programs.
- Create overloaded methods that have the same name but different types and/or numbers of parameters.
- See how `static` methods and fields are associated with classes rather than objects.
- Become familiar with key Java API packages, which contain groups of related classes.
- Learn how the method-call stack supports the method-call/return mechanism, and local variable creation and destruction.
- Understand argument promotion and casting.
- Use generative AI to leverage your Java learning experience.
- Continue our objects-natural approach with a case study on using objects of the Date/Time API classes to perform popular date and time manipulations.

Outline

5.1 Introduction

5.2 Declaring Methods

5.3 Notes on Declaring and Using Methods

5.4 Case Study: Die Rolling Simulation with Random-Number Generation

5.5 Case Study: A Game of Chance; Introducing `enum`s
5.6 Scope of Declarations
5.7 Method Overloading
5.8 Class `Math`: `static` Methods and Variables
5.9 Java API Packages
5.10 Method-Call Stack and Activation Records
5.11 Argument Promotion and Casting
5.12 Objects-Natural Case Study: Java Date/Time API
5.12.1 Data/Time Manipulations
5.12.2 Running the Program for a Different Locale Than Your Own
5.13 Wrap-Up
Special Section: Generative AI Exercises

5.1 Introduction

In this chapter, we introduce creating and calling custom methods. We'll use simulation techniques with random-number generation to program a version of a popular dice game that employs most of the programming techniques you've used to this point.

We'll demonstrate Java's `switch` enhancements. First, we'll use a standalone `switch` statement with more concise `case` labels. Then, we'll show `switch` expressions that yield values—these simplify code that declares a variable then uses a `switch` statement to assign a value to that variable.

We'll discuss scope, which determines where you can use the identifiers you've declared in a program. You'll see that classes can have multiple "overloaded" methods of the same name that perform similar tasks for different types and/or numbers of arguments.

We'll distinguish between methods you call on objects—such as `printf` on the `System.out` object—and `static` methods you call on a class rather than an object. We'll overview various packages from the Java API. You'll see how Java keeps track of which method is executing, how methods' local variables are maintained in memory, and how a method knows where to return after it completes execution. We discuss argument promotion—automatically converting an argument's value, if possible, to the type a method

expects to receive in its corresponding parameter—and casting, in which you explicitly force a conversion between types.

You'll continue using generative AI to leverage your Java learning experience. In the Objects-Natural Case Study, you'll use the Java Date/Time API to perform common date and time manipulations in an application that inputs your birth date and time (or approximations if you wish), then determines the day of the week on which you were born and how long you've been alive in years, months, days, hours, minutes and seconds.

5.2 Declaring Methods

So far, each class we declared contained one method named `main` and used various Java API features. Here, you'll declare then call your first custom method.

Class `MaximumFinder` (Fig. 5.1) has two methods—`main` (lines 6–22) and `maximum` (lines 25–39). The `maximum` method determines and returns the largest of three `double` values. Recall that the Java Virtual Machine (JVM) automatically calls `main` when you execute a program. Most methods do not get called automatically. You must call the `maximum` method explicitly to tell it to perform its task.

Click here to view code image

```
1  // Fig. 5.1: MaximumFinder.java
2  // Programmer-declared method maximum with three double parameters.
3  import java.util.Scanner;
4
5  public class MaximumFinder {
6     public static void main(String[] args) {
7        // create Scanner for input from command window
8        var input = new Scanner(System.in);
9
10       // prompt for and input three floating-point values
11       System.out.print(
12          "Enter three floating-point values separated by spaces: ");
13       double number1 = input.nextDouble(); // read first double
14       double number2 = input.nextDouble(); // read second double
15       double number3 = input.nextDouble(); // read third double
16
17       // determine the maximum value
18       double result = maximum(number1, number2, number3);
```

```
19
20          // display maximum value
21          System.out.printf("Maximum is: %.2f%n", result);
22       }
23
24       // returns the maximum of its three double parameters
25       public static double maximum(double x, double y, double z) {
26          double maximumValue = x; // assume x is the largest to start
27
28          // determine whether y is greater than maximumValue
29          if (y > maximumValue) {
30             maximumValue = y;
31          }
32
33          // determine whether z is greater than maximumValue
34          if (z > maximumValue) {
35             maximumValue = z;
36          }
37
38          return maximumValue;
39       }
40    }
```

```
Enter three floating-point values separated by spaces: 9.35 2.74 5.1
Maximum is: 9.35
```

```
Enter three floating-point values separated by spaces: 5.8 12.45 8.32
Maximum is: 12.45
```

```
Enter three floating-point values separated by spaces: 6.46 4.12 10.54
Maximum is: 10.54
```

Fig. 5.1 | Programmer-declared method `maximum` with three `double` parameters.

The `public` and `static` Keywords

Method `maximum`'s declaration (lines 25–39) begins with the `public` keyword to indicate that the method is "available to the public"—that is, it can be called from methods of other classes. The `static` keyword enables the `main` method (another `static` method) to call `maximum`, as shown in line 18, without qualifying the method name with the class name `MaximumFinder`—`static` methods in the same class can call each other directly. Any other class that uses `maximum` must fully qualify the method name, as in

Click here to view code image

```
MaximumFinder.maximum(10, 30, 20)
```

Many of our examples begin every method declaration with the `public` and `static` keywords. You'll learn about non-`public` and non-`static` methods in Chapter 8.

The Return Type

After `public` and `static` and before the method name is the **method's return type**, which specifies the type of data the method returns (that is, gives back to its caller) after performing its task. For example, when you go to an automated teller machine (ATM) and request your account balance, you expect the ATM to return your balance value. Similarly, when a statement calls method `maximum` in this example, the statement expects to receive back the largest of the three values. The return type `double` indicates that this method will perform a task then return a `double` value to the calling method—`main` in this example. You've already used methods that return information. For example, `Scanner`'s `nextInt` method inputs an integer entered by the user then returns that integer for use in the program. Sometimes, you'll define methods that perform a task but do not return any information. Such methods use the return type `void` (as in line 6 of `main`).

The Method Name and Parameters

The method name, `maximum`, follows the return type. Class names, method names and variable names are all identifiers. By convention, they use the camel-case naming scheme discussed in Chapter 2. Also, by convention, class names begin with an initial uppercase letter, and method and variable names begin with an initial lowercase letter.

The parentheses after the method name are required. A method that needs additional information to perform its task can specify one or more **parameters** to receive that information. Empty parentheses indicate that a method does not require additional information to perform its task.

You declare parameters as a comma-separated **parameter list** in the parentheses following the method name. Each parameter must specify a type and an identifier. The method `maximum` requires three `double` parameters named `x`, `y` and `z`.

Variables declared in a particular method's parameter list (such

as `x`, `y` and `z`) and body (such as `maximumValue`) are **local variables**—they can be used only in that method. Each method can access its own local variables but not those of other methods. Collectively, the elements in line 25 are often referred to as the **method header**.

Method `maximum`'s Body

The method header is always followed by the method's body, which is delimited by left and right braces, as in lines 25 and 39. A method's body contains one or more statements that perform the method's task.

We determine the maximum value as follows:

- Initially, we assume that parameter `x` contains the largest value, so line 26 declares the local variable `maximumValue` and initializes it with parameter `x`'s value.
- Of course, `y` or `z` may contain the largest value, so we must compare each with `maximumValue`. Lines 29–31 determine whether `y` is greater than `maximumValue`. If so, line 30 assigns `y` to `maximumValue`.
- Lines 34–36 determine whether `z` is greater than `maximumValue`. If so, line 35 assigns `z` to `maximumValue`.

At this point, `maximumValue` contains the largest value, so line 38 uses the **`return` statement** to return that value to line 18, where `maximum` was called. When a method returns to its caller, its local variables no longer exist in memory.

Local-Variable Type Inference

Line 8 in `main` introduces the **`var` keyword** for **local-variable type inference**. When you initialize a local variable in its declaration, the `var` keyword tells the compiler to infer (that is, determine) the variable's type from the type of its initializer. In line 8, we initialize `input` with a new `Scanner` object, so the compiler infers that `input`'s type is `Scanner`. This simplifies statements like

Click here to view code image

```
Scanner input = new Scanner(System.in);
```

that show the type on both sides of a variable declaration statement.

Getting the User Input and Calling Method `maximum`

Lines 11–15 in `main` prompt the user to enter three `double` values, then read them using **`Scanner`'s `nextDouble` method**, which returns the user's input as a `double` value. A method call supplies values—called arguments—for each of the method's parameters. For example, the argument to method `System.out.println` specifies the data to display. When we call `maximum` at line 18, its `x`, `y` and `z` parameters are initialized with copies of `number1`'s, `number2`'s and `number3`'s values, respectively. There must be one argument in the method call for each parameter in the method declaration. Also, each argument's type must be consistent with the corresponding parameter's type. For example, a parameter of type `double` can receive values like 7.35, 22 or –0.03456, but not `String`s like `"hello"` nor the `boolean` values `true` or `false`. Section 5.11 discusses the argument types you can provide in a method call for each primitive-type parameter. Line 21 displays the result.

Implementing Method `maximum` by Reusing Method `Math.max`

The entire body of our `maximum` method could be implemented concisely with two calls to `Math.max`, as follows:

Click here to view code image

```
return Math.max(x, Math.max(y, z));
```

The arguments to the first `Math.max` call are `x` and `Math.max(y, z)`. Before any method can be called, its arguments must be evaluated to determine their values. If an argument is a method call, the method call must be performed to determine its return value. So, in the preceding statement, `Math.max(y, z)` is evaluated first to determine the maximum of `y` and `z`. Then, that result is passed as the second argument to the other `Math.max` call, which returns the larger of its two arguments. This is a nice example of software reuse—we find the largest of three values by reusing `Math.max`, which finds the larger of two values.

Separating Programs into Smaller, More Manageable Pieces

Methods and classes allow you to separate a program's tasks into small self-contained units. Motivations for using methods and classes to create program components include:

- Software reuse. For example, in earlier programs, we did not have to define how to create and manipulate strings or read inputs from the user. Java provides these capabilities via the `String` and `Scanner` classes and their methods.
- Avoiding code repetition.
- Hiding complexity.
- Easier testing, debugging and maintenance.

To promote reusability, every method should perform a single, well-defined task, and the method's name should express that task effectively. We'll say lots more about software reusability in our treatment of object-oriented programming.

Generative AI

1 Prompt genAIs, asking them to find and correct the error(s) in the following code, which should return the result of adding the values of parameters `x` and `y`:

```
int sum(int x, int y) {
   int result = x + y;
}
```

Compare their responses with the following: The method is supposed to return an `int` but does not. To fix this, delete the variable `result` and place the statement

```
return x + y;
```

in the method, or add the following statement at the end of the method body:

```
return result;
```

5.3 Notes on Declaring and Using Methods

Calling Methods

There are three ways to call a method:

- Use a method name by itself to call another method of the same class—such as `maximum(number1, number2, number3)` in line 18 of Fig. 5.1.
- Use an object's name, followed by a dot (`.`) and the method name to call a non-`static` method of an object—such as `input.nextDouble()` in line 13 of Fig. 5.1, which calls a `Scanner` method from `MaximumFinder`'s `main` method. Non-`static` methods are typically called **instance methods**.
- Use the class name and a dot (`.`) to call a `static` method of a class—such as `Math.pow(1.0 + rate, year)` in Fig. 4.4.

Returning from Methods

There are three ways to immediately return control to the statement that calls a method:

- When the method-ending right brace is reached in a method with return type `void`.
- When the following statement executes in a method with return type `void`

  ```
  return;
  ```

- When a method returns a result with a statement of the following form in which the *expression* is evaluated and its result (and control) are returned to the caller:

  ```
  return expression;
  ```

Forgetting to return a value from a method that should return a value is a compilation error. If a method specifies a return type other than `void`, the method must contain a `return` statement that returns a value consistent with the method's return type. Returning a value from a method with a `void` return type is a compilation error.

5.4 Case Study: Die Rolling Simulation with Random-Number Generation

Let's take a brief diversion into simulation and game-playing. Here and in Section 5.5, we'll develop a game-playing program with multiple methods. As you'll see, the program uses most of the control statements presented thus far in the book and introduces several new programming concepts.

You can introduce the **element of chance** into a program via **`RandomGenerator` objects** from the **`java.util.random` package**. Such objects can produce random `boolean`, `byte`, `float`, `double`, `int`, `long` and other values using one of 13 random-number-generation algorithms listed at:

Click here to view code image

```
https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/
   util/random/package-summary.html
```

Creating a `RandomGenerator` Object

In the next several examples, we use `RandomGenerator` objects to produce random values using the default algorithm in the JDK.[1] The following statement uses **`RandomGenator` `static` method `getDefault`** to create a new `RandomGenerator` object:

Click here to view code image

```
RandomGenerator randomNumbers = RandomGenerator.getDefault()
```

1. L32X64MixRandom is the default algorithm used by the `RandomGenerator` object returned from `get-Default`. This could change in the future as random-number-generation algorithms improve. "Interface `RandomGenerator`." Accessed September 21, 2024. `https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/util/random/RandomGenerator.html`.

It can then be used to generate random values—we discuss only random `int` values here. For information on generating other random values, refer to

Click here to view code image

```
https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/
   util/random/RandomGenerator.html
```

Obtaining a Random `int` Value

Consider the following statement:

Click here to view code image

```
int randomValue = randomNumbers.nextInt();
```

`RandomGenerator` method `nextInt` generates a random `int` value from -2,147,483,648 to 2,147,483,647. If it truly produces values at random, then every value in that range should have an equal chance (or probability) of being chosen each time `nextInt` is called.

Changing the Range of Values Produced By `nextInt`

The range `nextInt` produces generally differs from the range a given Java program requires. For example, a program that simulates coin tossing might require only 0 for "heads" and 1 for "tails." A program that simulates the rolling a six-sided die might require random integers in the range 1–6. A program that randomly predicts the next type of spaceship (out of four possibilities) that will fly across the horizon in a video game might require random integers in the range 1–4. For cases like these, `RandomGenerator` provides other versions of `nextInt`:

- `nextInt` with one `int` argument returns a value from 0 up to, but not including, the argument's value. For example, to simulate a coin toss, the following statement returns 0 or 1:

  Click here to view code image

  ```
  int randomValue = randomNumbers.nextInt(2);
  ```

- `nextInt` with two `int` arguments returns a value from the first argument's value up to, but not including, the second argument's value. For example, to simulate rolling a six-sided die, the following statement returns a value in the range 1–6:

  Click here to view code image

  ```
  int randomValue = randomNumbers.nextInt(1, 7);
  ```

Rolling a Six-Sided Die 10 Times

Figure 5.2 shows two sample outputs confirming that the results of the preceding `nextInt` call are integers in the range 1–6 and that each run produces different numbers. Line 3 imports `RandomGenerator`. Line 8 creates the `RandomGenerator` object `randomNumbers` to produce random values. Line 13 executes 10 times in a loop to roll the die.

Click here to view code image

```
1  // Fig. 5.2: RandomIntegers.java
2  // Rolling a six-sided die.
3  import java.util.random.RandomGenerator;
4
5  public class RandomIntegers {
6     public static void main(String[] args) {
7        // randomNumbers object will produce random numbers
8        RandomGenerator randomNumbers = RandomGenerator.getDefault();
9
10       // loop 10 times
11       for (int counter = 1; counter <= 10; ++counter) {
12          // pick random integer from 1 to 6
13          int face = randomNumbers.nextInt(1, 7);
14
15          System.out.printf("%d  ", face); // display generated value
16    }
17
18       System.out.println();
19    }
20 }
```

```
1 1 3 6 4 2 5 5 3 4
```

```
3 6 3 5 3 2 6 6 4 5
```

Fig. 5.2 | Rolling a six-sided die.

Rolling a Six-Sided Die 60,000,000 Times

The numbers produced by `nextInt` occur with approximately equal likelihood. So, when simulating a six-sided die, we expect each face to occur about 16.667% (1/6th) of the time. For small numbers of die rolls, this is unlikely. For example, Fig. 5.2's second sample output contained no 1s. As we continue rolling a die many times, the "**law of large numbers**" says that each face's frequency should converge on 16.667% of the total die rolls. Let's demonstrate this by simulating 60,000,000 rolls of a die (Fig. 5.3).[2] Each face should appear approximately 10,000,000 times. Line 18 uses the `_` digit separator (Section 2.5) to make the `int` value `60_000_000` more readable.

2. When co-author Harvey Deitel first implemented this example in another language for his classes in 1976, he performed only 600 die rolls—6000 would have taken too long. On our current laptop test system, this program took about 1.33 seconds to complete 60,000,000 die rolls. 600,000,000 die rolls took 8.03 seconds. The die rolls occur

sequentially. Chapter 18, Concurrency: Platform Threads to Virtual Threads, explores how to parallelize applications to take advantage of today's multi-core computers.

Click here to view code image

```
// Fig. 5.3: RollDie.java
// Roll a six-sided die 60,000,000 times.
import java.util.random.RandomGenerator;

public class RollDie {
   public static void main(String[] args) {
      // randomNumbers object will produce random numbers
      RandomGenerator randomNumbers = RandomGenerator.getDefault();

      int frequency1 = 0; // count of 1s rolled
      int frequency2 = 0; // count of 2s rolled
      int frequency3 = 0; // count of 3s rolled
      int frequency4 = 0; // count of 4s rolled
      int frequency5 = 0; // count of 5s rolled
      int frequency6 = 0; // count of 6s rolled

      // tally counts for 60,000,000 rolls of a die
      for (int roll = 1; roll <= 60_000_000; ++roll) {
         int face = randomNumbers.nextInt(1, 7); // number from 1 to 6

         // use face value 1-6 to determine which counter to increment
         switch (face) {
            case 1 -> ++frequency1; // increment the 1s counter
            case 2 -> ++frequency2; // increment the 2s counter
            case 3 -> ++frequency3; // increment the 3s counter
            case 4 -> ++frequency4; // increment the 4s counter
            case 5 -> ++frequency5; // increment the 5s counter
            case 6 -> ++frequency6; // increment the 6s counter
         }
      }

      System.out.println("Face\tFrequency"); // output headers
      System.out.printf("1\t%d%n2\t%d%n3\t%d%n4\t%d%n5\t%d%n6\t%d%n",
         frequency1, frequency2, frequency3, frequency4,
         frequency5, frequency6);
   }
}
```

```
Face    Frequency
1       9996511
2       9999764
3       10001371
4       9997276
```

```
5       10003795
6       10001283
```

```
Face    Frequency
1       9999894
2       10000834
3       10000705
4       9997478
5       9999072
6       10002017
```

Fig. 5.3 | Roll a six-sided die 60,000,000 times.

As the sample outputs show, the values produced by `nextInt` enable the program to simulate rolling a six-sided die. The program uses nested control statements (a `switch` is nested inside the `for`) to determine the number of times each side of the die appears. The `for` statement (lines 18–30) iterates 60,000,000 times. During each iteration, line 19 produces a random value from 1 to 6. That value is then used as the controlling expression (line 22) in a `switch` statement that uses the new, more concise `case` label format (lines 22–29)—we'll discuss this in more detail momentarily. Based on the `face` value, the `switch` increments one of the six counter variables during each loop iteration. Run the program a couple of times and observe the different results during each run.

When we study arrays in Chapter 6, we'll show an elegant way to replace the entire `switch` in this program with a single statement. Then, when we study functional programming in Chapter 14, we'll show how to replace the loop that rolls the dice, the `switch` and the statement that displays the results with a single statement!

Introducing `switch` Statements with More Concise `case` Labels

The `switch` statement (introduced in Section 4.8) has several limitations:

- Its `case`s support only integer-type, `String` and `enum` constants. (We'll introduce `enum`s in Fig. 5.4).
- Each `case` label may specify only one value.
- If you do not include `break` statements, once a match occurs, `case`s fall through and continue executing subsequent cases

until the `switch` terminates—a possible logic error.

- The `default` case is optional—forgetting to include it could lead to logic errors.

Java enhanced `switch` statements and introduced **`switch` expressions** to improve on these limitations and provide additional capabilities. We present some of those improvements here and in Section 5.5, then show several more advanced features in Chapter 8.

A `switch` has several forms—we'll show two in this chapter:

- In this example, we use a `switch` statement (lines 22–29) with the new, more concise `case` labels.
- At the end of Section 5.5, we show a `switch` expression that yields a value, which can assign a value to a variable for use later in a program.

The `switch` statement in lines 22–29 differs from a traditional `switch` statement as follows:

- Rather than a `case` label that ends with a colon (`:`), the value tested in each `case` is followed by an **arrow (`->`) symbol** that points to the `case`'s body. By default, the body is one statement, but `case`s may contain a block of several statements in braces.
- After a `case` executes, the `switch` expression terminates immediately—no `break` statements are required.

Generalized Scaling and Shifting of Random Numbers

You can choose integers randomly from sets of values other than ranges of consecutive integers. For example, to obtain a random value from the sequence 2, 5, 8, 11 and 14, you could use the statement

Click here to view code image

```
int number = 2 + 3 * randomNumbers.nextInt(5);
```

In this case, `randomNumbers.nextInt(5)` produces values in the range 0–4. Each value produced is multiplied by 3 to produce a number in the sequence 0, 3, 6, 9 and 12. We add 2 to that value to shift the range of values and obtain a value from the sequence 2, 5,

8, 11 and 14. We can generalize this result as

Click here to view code image

```
int number = shiftingValue +
   differenceBetweenValues * randomNumbers.nextInt(scalingFactor);
```

where

- *shiftingValue* specifies the first number in the desired range of values,
- *differenceBetweenValues* represents the constant difference between consecutive numbers in the sequence and
- *scalingFactor* specifies how many numbers are in the range.

Secure Random Numbers

Some random-number-generation algorithms produce **deterministic random numbers**—that is, values that can be predicted by malicious programmers. Objects of the **`SecureRandom` class** (**`java.security` package**) produce **nondeterministic random numbers** that cannot be predicted. Deterministic random numbers have been the source of many software security breaches.[3] Most programming languages now have library features similar to `SecureRandom` for producing nondeterministic random numbers to help prevent such problems.

3. "Random number generator attack," Wikipedia. Wikimedia Foundation. Accessed August 30, 2024. https://en.wikipedia.org/wiki/Random_number_generator_attack.

You can create a secure random-number generator object as follows:

Click here to view code image

```
var randomNumbers = new SecureRandom();
```

`SecureRandom` has many of the same method names as `RandomGenerator`. Using `Secure-Random` instead of `RandomGenerator` to achieve higher levels of security incurs a significant performance penalty. For "casual" programs, prefer class `RandomGenerator`.

Generative AI

1 Prompt genAIs to write a Java statement using the

`RandomGenerator` object `random-Numbers` that produces a value from the set 6, 10, 14, 18 and 22 using only the `Random-Generator` object and arithmetic.

5.5 Case Study: A Game of Chance; Introducing `enum`s

A popular game of chance is the dice game known as craps, which is played in casinos and back alleys throughout the world. The rules of the game are straightforward:

> You roll two dice. Each die has six faces containing one, two, three, four, five and six spots, respectively. After the dice have come to rest, the sum of the spots on the two upward faces is calculated. You win if the sum is 7 or 11 on the first throw. You lose (i.e., the "house" wins) if the sum is 2, 3 or 12 on the first throw (called "craps"). If the sum is 4, 5, 6, 8, 9 or 10 on the first throw, that sum becomes your "point." To win, you must continue rolling the dice until you "make your point" (i.e., roll that same point value). You lose by rolling a 7 before making your point.

Figure 5.4 simulates the game of craps, using methods to implement the game's logic. The `main` method (lines 14–60) calls the `rollDice` method (lines 63–74) as necessary to roll the two dice and compute their sum. The four sample outputs show winning and losing on the first roll, and winning and losing on a subsequent roll.

Click here to view code image

```
1  // Fig. 5.4: Craps.java
2  // Craps class simulates the dice game craps.
3  import java.util.random.RandomGenerator;
4
5  public class Craps {
6     // get the default RandomGenerator for use in method rollDice
7     private static final RandomGenerator randomNumbers =
8        RandomGenerator.getDefault();
9
```

```
10      // enum type with constants that represent the game status
11      private enum Status {CONTINUE, WON, LOST};
12
13      // plays one game of craps
14      public static void main(String[] args) {
15         int myPoint = 0; // point if no win or loss on first roll
16         Status gameStatus; // can contain CONTINUE, WON or LOST
17
18         int sumOfDice = rollDice(); // first roll of the dice
19
20         // determine game status and point based on first roll
21         switch (sumOfDice) {
22            case 7:  // win with 7 on first roll
23            case 11: // win with 11 on first roll
24               gameStatus = Status.WON;
25               break;
26            case 2: // lose with 2 on first roll
27            case 3: // lose with 3 on first roll
28            case 12: // lose with 12 on first roll
29               gameStatus = Status.LOST;
30               break;
31            default: // did not win or lose, so remember point
32               gameStatus = Status.CONTINUE; // game is not over
33               myPoint = sumOfDice; // remember the point
34               System.out.printf("Point is %d%n", myPoint);
35               break;
36         }
37
38         // while game is not complete
39         while (gameStatus == Status.CONTINUE) { // not WON or LOST
40            sumOfDice = rollDice(); // roll dice again
41
42            // determine game status
43            if (sumOfDice == myPoint) { // win by making point
44               gameStatus = Status.WON;
45            }
46            else {
47               if (sumOfDice == 7) { // lose by rolling 7 before point
48                  gameStatus = Status.LOST;
49               }
50            }
51         }
52
53         // display won or lost message
54         if (gameStatus == Status.WON) {
55            System.out.println("Player wins");
56         }
57         else {
58            System.out.println("Player loses");
59         }
```

```
60      }
61
62      // roll dice, calculate sum and display results
63      public static int rollDice() {
64         // pick random die values
65         int die1 = randomNumbers.nextInt(1, 7); // first die roll
66         int die2 = randomNumbers.nextInt(1, 7); // second die roll
67
68         int sum = die1 + die2; // sum of die values
69
70         // display results of this roll
71         System.out.printf("Player rolled %d + %d = %d%n", die1, die2, sum);
72
73         return sum;
74      }
75   }
```

```
Player rolled 5 + 6 = 11
Player wins
```

```
Player rolled 1 + 2 = 3
Player loses
```

```
Player rolled 5 + 4 = 9
Point is 9
Player rolled 4 + 2 = 6
Player rolled 3 + 6 = 9
Player wins
```

```
Player rolled 2 + 6 = 8
Point is 8
Player rolled 5 + 1 = 6
Player rolled 2 + 1 = 3
Player rolled 1 + 6 = 7
Player loses
```

Fig. 5.4 | `Craps` class simulates the dice game craps.

Method `rollDice`

In the rules of the game, the player must roll two dice on each roll. The method `rollDice` (lines 63–74) rolls the two dice, computes and prints their sum, and returns their sum. The method is declared once, but called from lines 18 and 40 in `main`, which contains the

logic for one complete game of craps. Method `rollDice` takes no arguments, so it has an empty parameter list. Each time it's called, `rollDice` returns the sum of the dice, so the return type `int` is indicated in the method header (line 63). Although lines 65 and 66 look like the perform the same task, they do not necessarily produce the same result. Each statement produces a random value in the range 1–6. The variable `randomNumbers` (used in lines 65 and 66) is declared as a `private static final` variable of the class and initialized with a `RandomGenerator` object in lines 7–8:

- Making it `private` indicates it will be used only in class `Craps`.
- Making it a `static` class variable enables the class's `static` methods (like `roll-Dice`) to use it without having to create an object of the class.
- Making it `final` indicates that `randomNumbers` does not change after it's initialized. It's good practice to declare such variables `final` to prevent accidental changes.

Method `main`'s Local Variables

The game is reasonably involved. The player may win or lose on the first roll, or may win or lose on any subsequent roll. Method `main` (lines 14–60) uses

- local variable `myPoint` (line 15) to store the "point" if the player doesn't win or lose on the first roll,
- local variable `gameStatus` (line 16) to keep track of the overall game status and
- local variable `sumOfDice` (line 18) to store the most recent roll's sum of the dice.

Variable `myPoint` is initialized to `0` to ensure that the program will compile. If you do not initialize it, the compiler issues an error, because it is not assigned a value in every `case` of the `switch` statement—thus, the program could try to use `myPoint` before it's assigned a value. By contrast, `gameStatus` is assigned a value in every `case` and the `default` case of the `switch` statement—thus, it's guaranteed to be initialized before it's used. So, we do not need to initialize `gameStatus` in line 16.

`enum` Type `Status`

Local variable `gameStatus` (line 16) is declared to be of a new type called `Status` (declared at line 11). Type `Status` is a `private` member of the `Craps` class because `Status` will be used only in that class. `Status` is an **`enum` type**, which, in its simplest form, declares a set of constants represented by identifiers—that is, **named constants**. An `enum` type is a special class type that's introduced by the **`enum` keyword** and a type name (`Status`). Braces delimit an `enum` declaration's body. Inside the braces is a comma-separated list of **`enum` constants**, each representing a unique value. The identifiers in an `enum` must be unique and, by convention, use all uppercase letters. Using `enum` constants (like `Status.WON`, `Status.LOST` and `Status.CONTINUE`) rather than literal values (such as 0, 1 and 2) makes programs easier to read and maintain. You'll learn more about `enum` types in Chapter 8.

`Status` variables can be assigned only the three constants declared in the `enum` (line 11) or a compilation error will occur. When the game is won, the program sets local variable `gameStatus` to `Status.WON` (lines 24 and 44). When the game is lost, the program sets local variable `gameStatus` to `Status.LOST` (lines 29 and 48). Otherwise, the program sets local variable `gameStatus` to `Status.CONTINUE` (line 32) to indicate that the game is not over, and the dice must be rolled again.

Logic of the `main` Method

Line 18 in `main` calls `rollDice`, which picks two random values from 1 to 6, displays the values of the first die, the second die and their sum, and returns the sum. Method `main` next enters the `switch` statement (lines 21–36), which uses the `sumOfDice` value to determine whether the game has been won or lost or should continue with another roll. Lines 22–25 determine whether the player won on the first roll with `7` or `11`. Lines 26–30 determine whether the player lost on the first roll with `2`, `3` or `12`. After the first roll, if the game is not over, the `default` case (lines 31–35) sets `gameStatus` to `Status.CONTINUE`, saves `sumOfDice` in `myPoint` and displays the point.

If we're still trying to "make our point" (i.e., the game is continuing from a prior roll), lines 39–51 execute. Line 40 rolls the dice again. If `sumOfDice` matches `myPoint` (line 43), line 44 sets `gameStatus` to `Status.WON`, then the loop terminates because the game is complete. If `sumOfDice` is `7` (line 47), line 48 sets `gameStatus` to `Status.LOST`, and the loop terminates because the game is complete. When the game completes, lines 54–59 display a

message indicating whether the player won or lost, and the program terminates.

The program uses the various program-control mechanisms we've discussed. The `Craps` class uses two methods—`main` and `rollDice` (called twice from `main`)—and the `switch`, `while`, `if`...`else` and nested `if` control statements. Also, note the use of multiple `case` labels in the `switch` statement to execute the same statements for sums of `7` and `11` (lines 22–23) and for sums of `2`, `3` and `12` (lines 26–28).

`switch` Expressions Yield Values

Line 16's `gameStatus` declaration and the `switch` statement at lines 21–36 demonstrate a common Java code pattern: Declare a variable followed by a `switch` statement in which every branch assigns a value to that variable for use later in the code. A `switch` expression that yields a value simplifies this code pattern.

We can replace lines 16 and 21–36 with the following statement that initializes `gameStatus` with the result of a `switch` expression:

Click here to view code image

```
Status gameStatus = switch (sumOfDice) {
   case 7, 11 -> Status.WON;
   case 2, 3, 12 -> Status.LOST;
   default -> {
      myPoint = sumOfDice; // remember the point
      System.out.printf("Point is %d%n", myPoint);
      yield Status.CONTINUE; // game is not over
   }
}; // required semicolon to terminate the statement
```

In `switch` expressions, every branch must yield a value, you must have a `default` case and the statement in which the `switch` expression appears must end with a semicolon (`;`). The value each branch yields is specified to the right its arrow (`->`) symbol. This `switch` expression compares `sumOfDice` to each `case`. The first two `case`s

```
case 7, 11 -> Status.WON;
case 2, 3, 12 -> Status.LOST;
```

show that `switch` expressions can specify comma-separated lists of values:

- If the dice total `7` or `11`, the expression yields `Status.WON`.

- If the dice total `2`, `3` or `12`, the expression yields `Status.LOST`.

For all other sums, the `default` case executes a block of code, which

- assigns `sumOfDice` to `myPoint`,
- prints the point and
- yields the value `Status.CONTINUE` to initialize `gameStatus`.

When the code to the right of any branch's arrow (`->`) symbol is a block in braces (as in the `default` case), the block's last statement must be an explicit **`yield` statement** containing the value to yield. You may not use `return` statements in `switch` expressions.

We can also replace lines 43–50 in Fig. 5.4 with a `switch` expression that yields a value to update `gameStatus`'s value after subsequent die rolls:

Click here to view code image

```
// determine game status
gameStatus = switch (sumOfDice) {
   case 7 -> Status.LOST;
   default -> sumOfDice == myPoint ?
      Status.WON : Status.CONTINUE;
};
```

If `sumOfDice` is `7`, the `switch` expression yields `Status.LOST` as the new `gameStatus` value. For all other roll values, the `default` case uses a conditional expression to determine whether the sum of the dice is equal to `myPoint`. If so, the `default` case yields `Status.WON`; otherwise, it yields `Status.CONTINUE`. This `default` case does not require a `yield` statement because the code to the right of `->` is not a block of statements in braces.

Generative AI

1 Prompt genAIs to write a `switch` expression that yields `"heads"` or `"tails"` based on the value of the `int` variable `flip`, which contains `0` or `1`. For an invalid value, print a string containing the value and `"is invalid"`, then yield the empty string. Compare the Java code they generate to the following code:

Click here to view code image

```
String result = switch (flip) {
   case 0 -> "heads";
   case 1 -> "tails";
   default -> {
      System.out.printf("%d is invalid%n", flip);
      yield "";
   }
};
```

5.6 Scope of Declarations

You've seen declarations of various Java entities, such as classes, methods, variables and parameters. Declarations introduce names that can be used to refer to such Java entities. The **scope** of a declaration is the portion of the program that can refer to the declared entity by its name. Such an entity is said to be "in scope" for that portion of the program. This section introduces several important scope issues. The basic scope rules are as follows:

- The scope of a parameter declaration is the body of the method in which the declaration appears.
- The scope of a local variable declaration is from the point at which the declaration appears to the end of that block.
- The scope of a local variable declaration in a `for` statement's header is the body of the `for` statement and the other expressions in the header.
- A method or field's scope is the entire body of the class.

Any block may contain variable declarations. If a local variable or parameter in a method has the same name as a field of the class, the field is hidden until the block terminates execution—this is called **shadowing**. Although you can access a shadowed `static` class variable by preceding its name with the class's name and a dot (`.`), as in *ClassName*`.x`, avoiding shadowing can prevent subtle errors. It's a compilation error if multiple local variables have the same name in the same method.

Figure 5.5 demonstrates field and local-variable scopes. Line 6 declares and initializes the field `x` to `1`. This field is shadowed in any block (or method) that declares a local variable named `x`. Method `main` declares a local variable `x` (line 11) and initializes it

to `5`. Line 13 outputs this local variable's value to show that the field `x` (whose value is `1`) is shadowed in `main`.

Click here to view code image

```
1  // Fig. 5.5: Scope.java
2  // Scope class demonstrates field and local-variable scopes.
3
4  public class Scope {
5     // field that is accessible to all methods of this class
6     private static int x = 1;
7
8     // method main creates and initializes local variable x
9     // and calls methods useLocalVariable and useField
10    public static void main(String[] args) {
11       int x = 5; // method's local variable x shadows field x
12
13       System.out.printf("local x in main is %d%n", x);
14
15       useLocalVariable(); // useLocalVariable has local x
16       useField(); // useField uses class Scope's field x
17       useLocalVariable(); // useLocalVariable reinitializes local x
18       useField(); // class Scope's field x retains its value
19
20       System.out.printf("%nlocal x in main is %d%n", x);
21    }
22
23    // create and initialize local variable x during each call
24    public static void useLocalVariable() {
25       int x = 25; // initialized each time useLocalVariable is called
26
27       System.out.printf(
28          "%nlocal x on entering method useLocalVariable is %d%n", x);
29       ++x; // modifies this method's local variable x
30       System.out.printf(
31          "local x before exiting method useLocalVariable is %d%n", x);
32    }
33
34    // modify class Scope's field x during each call
35    public static void useField() {
36       System.out.printf(
37          "%nfield x on entering method useField is %d%n", x);
38       x *= 10; // modifies class Scope's field x
39       System.out.printf(
40          "field x before exiting method useField is %d%n", x);
41    }
42 }
```

```
local x in main is 5
```

```
local x on entering method useLocalVariable is 25
local x before exiting method useLocalVariable is 26

field x on entering method useField is 1
field x before exiting method useField is 10

local x on entering method useLocalVariable is 25
local x before exiting method useLocalVariable is 26

field x on entering method useField is 10
field x before exiting method useField is 100

local x in main is 5
```

Fig. 5.5 | `Scope` class demonstrates field and local-variable scopes.

The program declares two other methods—`useLocalVariable` (lines 24–32) and `useField` (lines 35–41)—that take no arguments and return no results. Method `main` calls each method twice (lines 15–18). Method `useLocalVariable` declares local variable `x` (line 25). When `useLocalVariable` is first called (line 15), it creates local variable `x` and initializes it to `25`, outputs the value of `x` (lines 27–28), increments `x` (line 29) and outputs the value of `x` again (lines 30–31). When `uselLocalVariable` is called a second time (line 17), it recreates local variable `x` and reinitializes it to `25`, so the output of each `useLocalVariable` call is identical.

Method `useField` does not declare any local variables. Therefore, when it refers to `x`, the class's field `x` (line 6) is used. When method `useField` is first called (line 16), it outputs the value (`1`) of field `x` (lines 36–37), multiplies the field `x` by `10` (line 38) and outputs the value (`10`) of field `x` again (lines 39–40) before returning. The next time method `useField` is called (line 18), the field has its modified value (`10`), so the method outputs `10`, then `100`.

Finally, in method `main`, the program outputs the value of local variable `x` again (line 20) to show that none of the method calls modified `main`'s local variable `x`, because the methods all referred to variables named `x` in other scopes.

Principle of Least Privilege

In a general sense, the **principle of least privilege** says that "things" should have the capabilities they need to get their job done, but no more. An example is the scope of a variable. A

variable should not be visible when it's not needed. Declare variables as close to where they're first used as possible.

Generative AI

1 Prompt genAIs with the Java code in Fig. 5.4 and ask them to explain the scope of every variable declared in the program.

5.7 Method Overloading

Methods of the same name can be declared in the same class if they have different sets of parameters (determined by the number, types and order of the parameters). This is called **method overloading**. When an overloaded method is called, the compiler selects the appropriate method based on the number, types and order of the arguments in the call. Method overloading is commonly used to create several methods with the same name that perform the same or similar tasks but on different types and/or different numbers of arguments. For example, the `Math` class provides method `min` that is overloaded with four versions each:

- One that returns the smaller of two `double` parameters.
- One that returns the smaller of two `float` parameters.
- One that returns the smaller of two `int` parameters.
- One that returns the smaller of two `long` parameters.

Our next example demonstrates declaring and invoking overloaded methods.

Declaring Overloaded Methods

Class `MethodOverload` (Fig. 5.6) declares overloaded `square` methods—one calculates the square of an `int` (and returns an `int`), and one calculates the square of a `double` (and returns a `double`). Although these methods have the same name and similar parameter lists and bodies, it may help to think of them as different methods named "`square` of `int`" and "`square` of `double`," respectively.

Click here to view code image

```
1  // Fig. 5.6: MethodOverload.java
2  // Overloaded method declarations.
3
4  public class MethodOverload {
5     // test overloaded square methods
6     public static void main(String[] args) {
7        System.out.printf("Square of integer 7 is %d%n", square(7));
8        System.out.printf("Square of double 7.5 is %f%n", square(7.5));
9     }
10
11    // square method with int argument
12    public static int square(int intValue) {
13       System.out.printf("%nCalled square with int argument: %d%n",
14          intValue);
15       return intValue * intValue;
16    }
17
18    // square method with double argument
19    public static double square(double doubleValue) {
20       System.out.printf("%nCalled square with double argument: %f%n",
21          doubleValue);
22       return doubleValue * doubleValue;
23    }
24 }
```

```
Called square with int argument: 7
Square of integer 7 is 49

Called square with double argument: 7.500000
Square of double 7.5 is 56.250000
```

Fig. 5.6 | Overloaded method declarations.

Line 7 invokes method `square` with the argument `7`. Literal integer values are treated as type `int`, so the method call in line 7 invokes the version of `square` at lines 12–16 that specifies an `int` parameter. Similarly, line 8 invokes method `square` with the argument `7.5`. Literal floating-point values are treated as type `double`, so the method call in line 8 invokes the version of `square` at lines 19–23 that specifies a `double` parameter. Each method first outputs a line of text to show that the proper method was called in each case. The values in lines 8 and 20 are displayed with the `%f` format specifier, which outputs values of type `float` or `double` with six digits of precision to the right of the decimal point by default.

Distinguishing Between Overloaded Methods

The compiler distinguishes overloaded methods by their **signatures**—a combination of the method's name and the number, types and order of its parameters, but not its return type. If the compiler looked only at method names during compilation, the code in Fig. 5.6 would be ambiguous—the compiler would not know how to distinguish between the two `square` methods. Internally, the compiler uses longer method names that include the original method name, the types of each parameter and the exact order of the parameters to determine whether the methods in a class are unique in that class.

For example, in Fig. 5.6, the compiler might (internally) use the logical name "`square` of `int`" for the `square` method that specifies an `int` parameter and "`square` of `double`" for the `square` method that specifies a `double` parameter—the actual names the compiler uses are messier. If `method1`'s declaration begins as

```
void method1(int a, float b)
```

then the compiler might use the logical name "`method1` of `int` and `float`." If the parameters are specified as

```
void method1(float a, int b)
```

then the compiler might use the logical name "`method1` of `float` and `int`." The order of the parameter types is important—the compiler considers the preceding two `method1` headers to be distinct.

Return Types of Overloaded Methods

In discussing the logical names of methods used by the compiler, we did not mention return types. Method calls cannot be distinguished only by return type. When two methods have the same signature and different return types, the compiler issues an error message indicating that the method is already defined in the class. Overloaded methods can have different return types if the methods have different parameter lists. Also, overloaded methods need not have the same number of parameters. Declaring overloaded methods with identical parameter lists is a compilation error.

Generative AI

1 Prompt genAIs with the following code and ask them which method would be called if you pass to `multiply` the values `7` and

2.2:

Click here to view code image

```
public static double multiply(double x, double y) {
   return x * y;
}

public static int multiply(int x, int y) {
   return x * y;
}
```

Ask them to write a Java program that proves which `multiply` method is called, then execute the program to confirm the result.

5.8 Class `Math`: `static` Methods and Variables

Section 3.12 discussed that most methods are called on specific objects and that some classes contain convenient `static` methods to perform common tasks. Such methods are also known as **class methods**. Recall that we used class `Math`'s `static` method `pow` to raise a value to a power in Fig. 4.4. For any class with `static` methods that's imported into your program, you can call the `static` methods by specifying the class's name, followed by a dot (`.`) and the method name, as in

ClassName`.`*methodName*`(`*arguments*`)`

`Math` Class Methods

We use `Math` class methods here to present the concept of `static` methods. Recall that `Math` is part of the `java.lang` package, which is implicitly imported by the compiler, so can use class `Math`'s methods without explicitly importing the class.

Class `Math` provides `static` methods that perform common mathematical calculations. For example, you can calculate the square root of `900.0` with the `static` method call

```
Math.sqrt(900.0)
```

This expression evaluates to `30.0`. Method `sqrt` takes an argument of type `double` and returns a result of type `double`. To output the value of the preceding method call, you might write the statement

Click here to view code image

```
System.out.println(Math.sqrt(900.0));
```

Here, the value `sqrt` returns becomes the argument to `println`. The `Math` class's methods are all `static`, so you do not need to create a `Math` object to call `sqrt`. You call each `Math` method by preceding its name with the class name `Math` and the dot (`.`) separator.

Method arguments may be constants, variables or expressions. If `c = 13.0`, `d = 3.0` and `f = 4.0`, then the statement

Click here to view code image

```
System.out.println(Math.sqrt(c + d * f));
```

calculates and prints the square root of `13.0 + 3.0 * 4.0 = 25.0`—namely, `5.0`. The following table summarizes some `Math` class `static` methods—*x* and *y* in the table have type `double`. Methods `abs`, `max` and `min` are overloaded for `double`, `float`, `long` or `int` parameters.

Method	Description
`abs(x)`	absolute value of *x*
`abs(0.0)` is `0.0`	
`abs(-23.7)` is `23.7`	
`ceil(x)`	rounds *x* to the smallest integer not less than *x*
`ceil(-9.8)` is `-9.0`	
`cos(x)`	trigonometric cosine of *x* (*x* in radians)
`exp(x)`	exponential method e^x
`exp(2.0)` is `7.38906`	
`floor(x)`	rounds *x* to the largest integer not greater than *x*
`floor(-9.8)` is `-10.0`	
`log(x)`	natural logarithm of *x* (base *e*)
`log(Math.E * Math.E)` is `2.0`	
`max(x, y)`	larger value of *x* and *y*
`max(-2.3, -12.7)` is `-2.3`	
`min(x, y)`	smaller value of *x* and *y*
`min(-2.3, -12.7)` is `-12.7`	
`pow(x, y)`	*x* raised to the power *y* (i.e., x^y)
`pow(9.0, 0.5)` is `3.0`	

```
trigonometric sine of x (x in radians)
square root of x
trigonometric tangent of x (x in radians)
```

Class Variables and `Math` Class `static` Constants `PI` and `E`

Just as a class may contain `static` methods, a class also may contain `static` variables that belong to the class, not to specific objects of the class. Such variables are also known as **class variables**. Class `Math` declares the constants `Math.PI` and `Math.E`. Each is a `double` approximation of a commonly used mathematical constant with 15 digits to the right of the decimal point:

- **`Math.PI`** (3.141592653589793) is the ratio of a circle's circumference to its diameter.
- **`Math.E`** (2.718281828459045) is the base value for natural logarithms (calculated with class `Math`'s `static` method `log`).

These constants are declared in class `Math` with the modifiers `public`, `final` and `static`:

- Making them `public` allows you to use them in your own classes.
- Any variable declared with the **`final` keyword** is constant—its value cannot change after the variable is initialized.
- Making these variables `static` allows them to be accessed via the class name `Math` and a dot (`.`) separator, just as class `Math`'s methods are.

A class's variables are sometimes called **fields**. Section 8.16 says more about `static` fields.

Why Is Method `main` Declared `static`?

When you execute the Java Virtual Machine (JVM) with the `java` command, the JVM attempts to invoke the `main` method of the class you specify. Declaring `main` as `static` allows the JVM to invoke `main` without creating an object of the class. When you execute your program, you specify its class name as an argument to the `java` command, as in

Click here to view code image

```
java ClassName argument1 argument2 ...
```

The JVM loads the class specified by *ClassName* and uses that class name to invoke the `main` method. In the preceding command, *ClassName* is an argument to the JVM that tells it which class to execute. Following the *ClassName*, you can also specify a list of `String`s (separated by spaces) as arguments that the JVM will pass to your program—these are known as **command-line arguments**. Such arguments might specify options required to run the program, such as the name of a file a program should process. Every class may contain `main`—only the `main` of the class used to execute the program is called. As you'll learn in Chapter 6, Arrays and `ArrayList`s, your program can access those command-line arguments and use them to customize the program.

Generative AI

1 Prompt genAIs, asking what the pros and cons are for including a `main` method in every Java class.

5.9 Java API Packages

You typically write programs by combining

- prepackaged methods and classes available in the Java Application Programming Interface (Java API),
- methods and classes available in a vast number of open-source and proprietary third-party libraries, and
- new methods and classes you and your colleagues write.

A great strength of Java is the Java API's nearly 5000 predefined classes and other types, which are grouped into 200+ packages for `import` into your programs. You should familiarize yourself with these predefined capabilities so you can reuse them rather than "reinventing the wheel." The following table shows some of the Java API packages used in this book:

Package	Description
`java.io`	The **Java Input/Output Package** contains classes and other

types that enable programs to input and output data. (See Chapter 11, Files, I/O Streams, JSON Serialization & CSV Files.)

The **Java Language Package** (imported into all programs by default) contains core capabilities required by many Java programs (discussed throughout the book). This package includes class `Thread`, which contains support for Java powerful new virtual threads. Parallel applications can create massive numbers of virtual threads to represent concurrent activities. We'll discuss virtual threads in Chapter 18, Concurrency: Platform Threads to Virtual Threads.

The **Java Math Package** provides capabilities for performing calculations with integers of any size (shown in our discussion of class `BigInteger` in Section 3.12) and floating-point numbers that are represented exactly (shown in our discussion of class `BigDecimal` in Section 4.11).

The **Java Security Package** contains classes and other types for enhancing security, such as `SecureRandom` (mentioned in Section 5.4).

The **JDBC Package** contains classes and other types for working with databases. (Refer to Chapter 20, Accessing Databases with JDBC and SQLite.)

The **Java Text Package** provides capabilities for manipulating text, dates, times and numbers independent of natural languages like English, Spanish, Chinese, Hindi, etc., enabling programs used internationally to handle text, dates, times and numbers using the local formats users expect. (We use features of this package in the JavaFX chapters.)

The **Date/Time API Package** contains classes and other types for working with dates and times. (See Section 5.12, Objects-Natural Case Study: Java Date/Time API and Chapter 18, Concurrency: Platform Threads to Virtual Threads.)

The **Java Utilities Package** contains utility classes and other types for storing and processing large amounts of data, formatting text and more. (See Chapter 12, Generic Collections.)

The **Java Concurrency Package** contains utility classes and other types for implementing programs that can perform multiple tasks in parallel, taking advantage of today's multi-core processors. (See Chapter 18, Concurrency: Platform Threads to Virtual Threads.)

The **Java Regular Expressions Package** provides capabilities for locating, replacing and validating patterns in text. (See Chapter 7, Strings, NLP and Regex: Generative AI Foundations.)

These packages contain classes and other types for functional

programming. (See Chapter 14, Functional Programming with Lambdas & Streams.)

The **Java Cryptography Architecture (JCA) Package** provides capabilities for protecting data with encryption and decryption using popular algorithms. (See Chapter 7, Strings, NLP and Regex: Generative AI Foundations, and Chapter 11, Files, I/O Streams, JSON Serialization & CSV Files, for our objects-natural case studies on private-key and public-key cryptography.)

Related packages are grouped into another program unit called a module.[4] The Java API's 200+ packages are divided into 58 modules. Most of the packages in the preceding table are in the **`java.base` module**. For an overview of this module's packages, visit

4. You'll learn how to group custom classes into packages and work with modules in Chapter 21, Java Platform Module System.

Click here to view code image

```
https://docs.oracle.com/en/java/javase/23/docs/api/java.base/module-
   summary.html
```

In the packages table at the preceding web page, you can click a package name to learn more about it. The Index link at the top of the page, contains an alphabetical listing of every Java API type, method and constant. For more information on using the Java API documentation, refer to

Click here to view code image

```
https://docs.oracle.com/en/java/javase/23/docs/api/help-doc.html
```

Generative AI

1 Prompt genAIs to create a ranked list of the top Java API packages that Java programmers will want to become familiar with.

5.10 Method-Call Stack and Activation Records

To understand how Java performs method calls, we first consider a

data structure (i.e., collection of related data items) known as a **stack**. Think of a stack as analogous to a pile of dishes. When a dish is placed on the pile, it's placed at the top—referred to as **pushing** the dish onto the stack. Similarly, when a dish is removed from the pile, it's removed from the top—referred to as **popping** the dish off the stack. Stacks are known as **last-in, first-out (LIFO) data structures**—the last item pushed (inserted) on the stack is the first item popped (removed) from the stack.

Method-Call Stack

The **method-call stack** (also known as the **program-execution stack**) supports the method call/return mechanism. It also supports the creation, maintenance and destruction of each called method's local variables. As we'll see in this section's diagrams, last-in, first-out (LIFO) behavior is precisely what a method needs to return to its caller.

Stack Frames

As each method is called, it may, in turn, call other methods, which may, in turn, call other methods—all before any of the methods return. Each method eventually must return control to the method that called it. So, somehow, the system must keep track of the return addresses. The method-call stack is the perfect data structure for handling this information. When a method calls another method, an entry is pushed onto the stack. This entry—called a **stack frame** or an **activation record**—contains the return address that the called method needs to return to the calling method. It also includes some additional information we'll soon discuss. When the called method returns to its calling method, the stack frame for the called method is popped, and control transfers to the return address in the popped stack frame.

The beauty of the call stack is that each called method always finds at the top of the call stack the information it needs to return to its caller. If a method calls another method, a stack frame for the new method call is simply pushed onto the call stack. Thus, the return address required by the newly called method to return to its caller is now at the top of the stack.

Local Variables and Stack Frames

The stack frames have another important responsibility. Most methods have local variables, which include the parameters and

any local variables the method declares. Non-`static` local variables need to exist while a method is executing. They must remain active if the method makes calls to other methods. But when a called method returns to its caller, the called method's local variables need to "go away." The called method's stack frame is a perfect place to reserve the memory for the called method's local variables. That stack frame exists as long as the called method is active. When that method returns, its stack frame is popped from the stack, and its local variables no longer exist.

Stack Overflow

Of course, a computer's memory is finite, so only a certain amount of memory can be used to store activation records on the method-call stack. If more method calls occur than can have their activation records stored on the method-call stack, a fatal error known as **stack overflow** occurs.[5]

5. This is how the website `stackoverflow.com` got its name. This is an excellent website for getting answers to your programming questions.

Method-Call Stack in Action

Now, let's consider how the call stack supports the operation of a `square` method (lines 13–15) called by `main` (lines 6–10 of Fig. 5.7).

Click here to view code image

```
1  // Fig. 5.7: SquareDemo.java
2  // Square method used to demonstrate the
3  // method-call stack and activation records.
4
5  public class SquareDemo {
6     public static void main(String[] args) {
7        int y = 10; // value to square (local variable in main)
8
9        int result = square(y); // calculate y squared and store in result
10    }
11
12    // returns the square of an integer
13    public static int square(int x) {
14       return x * x; // calculate square and return result
15    }
16 }
```

Fig. 5.7 | `square` method used to demonstrate the method-call stack and activation records.

Step 1: Java Virtual Machine Invokes `main` to Execute the Application

First, the JVM calls `main`. This pushes a stack frame onto the stack, as shown in the following diagram. The stack frame tells `main` how to return to the JVM (that is, transfer to return address `R1`) and contains the space for `main`'s local variable `y`, which is initialized to 10, and local variable result, which is not yet initialized. (To keep these diagrams concise, we do not show the `public` and `static` keywords for each method in this example.)

Step 2: `main` Invokes Method `square` to Perform the Calculation

Before returning to the JVM, `main` calls method `square` in line 9 of Fig. 5.7. This pushes a stack frame for `square` (lines 13–15) onto the method-call stack:
This stack frame contains the address `R2` that `square` needs to return to `main` and the memory for `square`'s parameter `x`.

Step 3: `square` Returns Its Result to `main`

After `square` calculates the square of its argument, it needs to return to `main`—and no longer needs the memory for its variable `x`. So the stack is popped, giving `square` the return location `R2` in `main` and losing `square`'s local variable. The following diagram shows the method-call stack after `square`'s stack frame has been popped and `square`'s return value has been stored in `main`'s local variable `result`:
Reaching the closing right brace of `main` (line 10 of Fig. 5.7) pops its stack frame from the stack. This gives `main` the address it needs to return to the JVM (`R1` in the first diagram). At this point, `main`'s local variables, `y` and `result`, are no longer available.[6]

6. You may want to check out `https://cscircles.cemc.uwaterloo.ca/java_visualize/`, which creates interactive visualizations of the method call/return mechanism, based on code you provide.

Data Structures

You've now seen how valuable the stack data structure is in supporting program execution. Data structures have many important applications in computer science. Chapter 6 introduces Java's built-in array data structure and explores the Java API's `ArrayList` collection class in the chapter's Objects-Natural Case

Study. We discuss many of Java's other data structures (collections) in more detail in Chapter 12, Generic Collections.

Generative AI

1 Prompt genAIs with Fig. 5.1's Java code. Ask them to explain how the method-call stack works for that code. Tell them to use text-only, step-by-step diagrams made of characters.

5.11 Argument Promotion and Casting

Another important feature of method calls is **argument promotion**—converting an argument's value, if possible, to the type the method expects to receive in its corresponding parameter. For example, a program can call `Math` method `sqrt` with an `int` argument even though a `double` argument is expected. The statement

Click here to view code image

```
System.out.println(Math.sqrt(4));
```

correctly evaluates `Math.sqrt(4)` and prints the value `2.0`. The method declaration's parameter list causes a copy of the `int` value `4` to be converted to the `double` value `4.0` before the value is passed to method `sqrt`. Conversions that do not adhere to Java's **promotion rules** may lead to compilation errors. These rules specify which conversions are allowed and can be performed without losing data. In the `sqrt` example above, an `int` is converted to a `double` without changing its value. However, converting a `double` to an `int` truncates the fractional part of the `double` value. Thus, part of the value is lost. Converting large integer types to small integer types (for example, `long` to `int` or `int` to `short`) may also change values.

The promotion rules apply to expressions containing values of two or more primitive types and primitive-type values passed to methods. A temporary copy of each value is promoted to the "highest" type—the types of the original values remain unchanged. The following table lists the primitive types and the types to which each can be promoted:

Type	Valid promotions
`double`	

None
`double`
`float` or `double`
`long`, `float` or `double`
`int`, `long`, `float` or `double`
`short`, `int`, `long`, `float` or `double` (but not `char`)
`byte`, `short`, `int`, `long`, `float` or `double` (but not `char`)
None (`boolean` values are not numbers in Java)

The valid promotions for a given type are always to a type higher in the table. For example, an `int` can be promoted to the higher types `long`, `float` or `double`.

Converting values to types lower in the preceding table will result in different values if the lower type cannot represent the value of the higher type (for example, the `int` value 2000000 cannot be represented as a `short`, and any floating-point number with digits after its decimal point cannot be represented in an integer type such as `long`, `int` or `short`). Therefore, in cases where information could be lost due to conversion, the compiler requires you to use a cast operator (introduced in Section 3.7) to explicitly force the conversion to occur—otherwise, a compilation error occurs. This enables you to "take control" from the compiler. You essentially say, "I know this conversion might cause a loss of information, but for my purposes here, that's fine."

Casting a primitive-type value to another primitive type may change the value if the new type is not a valid promotion. For example, casting a floating-point value to an integer value may introduce truncation errors (loss of the fractional part) into the result.

Suppose method `square` calculates the square of an integer and thus requires an `int` argument. To call `square` with a `double` argument named `doubleValue`, we would be required to write the method call as

```
square((int) doubleValue)
```

This method call explicitly casts `doubleValue`'s value to a a temporary `int` for use in `square`. So, if `doubleValue`'s value is `4.5`, the method receives `4` and returns `16`, not `20.25`.

Generative AI

1 Prompt genAIs with the allowed Java primitive-type promotions discussed in this section and ask why `char` is higher in the table than `short`.

5.12 Objects-Natural Case Study: Java Date/Time API

Let's introduce features of the **Java Date/Time API**, which provides robust capabilities for

- conveniently managing date and time information,
- performing date and time calculations and
- presenting dates and times using **locale-specific formatting** that considers the user's spoken (or written) language, country and time zone (that is, the user's **locale**).

We'll build a program that inputs a user's birth date and time (or approximations if they wish), then determines and displays

- the day of the week the user was born,
- the number of years, months and days the user has been alive, and
- the total months, days, hours, minutes and seconds the user has been alive.

Here is a sample output for a person born July 20, 1969, at 10:56 PM Eastern Time—your output will differ since you did not run the program on the same date and time as we did:[7]

7. This is the day and time Neil Armstrong stepped onto the moon. "Apollo 11 Timeline," Accessed August 30, 2024. https://airandspace.si.edu/explore/stories/apollo-missions/apollo-11-moon-landing/apollo-11-timeline.

Click here to view code image

```
Enter your birthday (yyyy-MM-dd): 1969-07-20
Enter your birth time (HH:mm): 22:56

Birth date/time: Jul 20, 1969, 10:56:00 PM
Current date/time: Jun 15, 2024, 12:40:56 PM

You were born on a Sunday
You have been alive for 54 years, 10 months, and 26 days
```

```
Total months:  658
Total days:    20,054
Total hours:   481,285
Total minutes: 28,877,144
Total seconds: 1,732,628,696
```

In Section 5.12.2, we'll show how to run the program with command-line arguments that specify spoken (or written) languages and locales so you can obtain locale-specific formatting of the dates, times and numeric values displayed in the preceding output. For simplicity, we will not translate the other English text in this example.

Some key Date/Time API types we'll use in this case study include:

- The **`LocalDate` class**, **`LocalTime` class**, and **`LocalDateTime` class** (each from the **`java.time` package**[8]) represent dates, times and combined date-time values based on the user's local date and time.

  8. "Package `java.time`," Accessed September 21, 2024. https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/time/package-summary.html.

- The **`Period` class** and **`Duration` class** (each from the `java.time` package) represent periods of time. A `Period` is date-based, storing years, months and days. A `Duration` is time-based, storing hours, minutes and seconds.
- The **`DateTimeFormatter` class**, **`FormatStyle` enum** and **`TextStyle` enum** (each from the **`java.time.format` package**[9]) enable precise date and time formatting.

  9. "Package `java.time.format`," Accessed September 21, 2024. https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/time/format/package-summary.html.

- The **`ChronoUnit` enum** (from the **`java.time.temporal` package**[10]) provides constants for specifying the units of values returned by various Date/Time classes' calculation methods.

  10. "Package `java.time.temporal`," Accessed September 21, 2024. https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/time/temporal/package-summary.html.

The beauty of the objects-natural approach is that the types discussed here consist of nearly 4000 source-code lines (not

including comments). Implementing these types yourself would take significant time to write, test, debug and prove them correct. But, you can benefit from their powerful capabilities simply by writing small numbers of statements that manipulate objects of these types and call their methods.

5.12.1 Data/Time Manipulations

Figure 5.8 implements the application using various Date/Time API features. We've split the program into parts for discussion purposes.

Importing the Java API Classes and `enum`s

Lines 4–14 `import` the classes and `enum`s used in this program.

Click here to view code image

```
1  // Fig. 5.8: AgeCalculator.java
2  // Using the Date/Time API to calculate how long the user has
3  // been alive and determine the day of the week on which they were born.
4  import java.time.Duration;
5  import java.time.LocalDate;
6  import java.time.LocalDateTime;
7  import java.time.LocalTime;
8  import java.time.Period;
9  import java.time.format.DateTimeFormatter;
10 import java.time.format.FormatStyle;
11 import java.time.format.TextStyle;
12 import java.time.temporal.ChronoUnit;
13 import java.util.Locale;
14 import java.util.Scanner;
15
```

Fig. 5.8 | Using the Date/Time API to calculate how long the user has been alive and determine the day of the week on which they were born.

Inputting the Birthday and Birth Time from the User

Lines 21–26 prompt for and input the user's birthday and birth time:

Click here to view code image

```
16 public class AgeCalculator {
17    public static void main(String[] args) {
```

```
18          var input = new Scanner(System.in);
19
20          // input user's birthday and convert it to a LocalDate
21          System.out.print("Enter your birthday (yyyy-MM-dd): ");
22          var birthday = LocalDate.parse(input.nextLine());
23
24          // input user's birth time and convert it to a LocalTime
25          System.out.print("Enter your birth time (HH:mm): ");
26          var birthTime = LocalTime.parse(input.nextLine());
27
```

```
Enter your birthday (yyyy-MM-dd): 1969-07-20
Enter your birth time (HH:mm): 22:56
```

The code operates as follows:

- Line 21 specifies that the birthday should be input in the format `yyyy-MM-dd`, which represents a four-digit year (e.g., 1969), a two-digit month (e.g., 07) and a two-digit day (e.g., 20), each separate from the next by a dash (`-`). The Date/Time API uses capital `M`'s to represent months and lowercase `m`'s to represent minutes. The `DateTimeFormatter` class's documentation contains a table explaining the letters used in date and time formats.[11]

  11. "Class `DateTimeFormatter`," Accessed September 21, 2024. https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/time/format/DateTimeFormatter.html.

- Line 22 uses the **`Scanner` class's `nextLine` method** to read a line of text as a string. It then passes the string to the **`LocalDate` class's `parse` method**, which converts the date to a `LocalDate` object that we store in `birthday`.
- Line 25 specifies that the birth time should be input in the `HH:mm` format, representing a two-digit hour and a two-digit minute, separated by a colon (`:`). The Date/Time API uses capital `H`'s to represent hours in 24-hour clock format, whereas lowercase `h`'s represent 12-hour clock format.
- Line 26 reads a line of text as a string and passes it to the **`LocalTime` class's `parse` method** to convert the time to a `LocalTime` object, then stores it in `birthTime`.

Creating `LocalDateTime` Objects for the Birthday and Birth Time and the Current Date and Time

To prepare for date/time calculations, line 29 converts the `birthday` and `birthTime` objects into a `LocalDateTime` object by calling the **`LocalDateTime` class's `of` method** and storing the result in `birthDateTime`. We'll calculate differences between the current date and time and `birthDateTime`, so line 32 calls the **`LocalDateTime` class's `now` method** to get the current date and time on the user's system and store it in `now`. Line 35 calls our `displayDatesAndTimes` method (discussed shortly) to display the `LocalDateTime` objects `birthDateTime` and `now` in the default format for the user's locale.

Click here to view code image

```
28          // combine the birthday and birthTime into a LocalDateTime object
29          var birthDateTime = LocalDateTime.of(birthday, birthTime);
30
31          // get the current date and time as a LocalDateTime object
32          var now = LocalDateTime.now();
33
34          // show the dates/times
35          displayDatesAndTimes(birthDateTime, now);
36
```

```
Birth date/time: Jul 20, 1969, 10:56:00 AM
Current date/time: Jun 15, 2024, 11:40:59 AM
```

Calculating the `Period` and `Duration` Between the `LocalDateTime` Objects

Lines 38 and 41 calculate the difference between the birth date/time and the current date/time in years, months and days, then hours, minutes, and seconds:

Click here to view code image

```
37          // calculate date/time difference in years, months and days
38          var period = Period.between(birthday, now.toLocalDate());
39
40          // calculate date/time difference in hours, minutes and seconds
41          var duration = Duration.between(birthDateTime, now);
42
```

Lines 38 and 41 use methods of the `Period` and `Duration` classes:

- The **`Period` class's `static` method `between`** (line 38) returns

a `Period` object representing the difference between two `LocalDate` objects in years, months and days. Here, we pass the `birthday` object created in line 22 and the `now` object's `LocalDate` representation returned by the **`LocalDateTime` class's `static` `toLocalDate` method**. This method ignores a `LocalDateTime`'s time part.

- **`Duration` class's `static` method `between`** (line 41) returns a `Duration` object representing the difference between two `LocalDateTime` objects (`birthDateTime` and `now`) in hours, minutes and seconds. This calculation uses the full dates and times.

Determining a Date's Day of the Week

Lines 44–46 display the day of the week on which the user was born:

Click here to view code image

```
43          // display day of the week the user was born
44          System.out.printf("You were born on a %s%n%n",
45             birthday.getDayOfWeek().getDisplayName(
46                TextStyle.FULL, Locale.getDefault()));
47
```

```
You were born on a Sunday
```

The **`LocalDate` class's `getDayOfWeek` method** (line 45) returns a constant from the `DayOfWeek` `enum`, which contains constants for all seven days. Each day has a human-readable, locale-specific string representation returned by the **`DayOfWeek` `enum`'s `getDisplayName` method**, which has two arguments:

- The **`TextStyle.FULL`**[12] constant (line 46) specifies that the full name of the day should be returned (e.g., "`Monday`", "`Tuesday`").

  12. "Enum Class `TextStyle`," Accessed September 21, 2024. https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/time/format/TextStyle.html.

- The second argument specifies the locale to use for formatting. The **`Locale` class's `static` `getDefault` method** (line 46) returns the default `Locale` object with settings specific to the user's system.

Displaying How Long the User Has Been Alive

Lines 49–61 display how long the user has been alive in years, months and days, then in total months, days, hours, minutes and seconds—here we define `printf`'s format string as a text block that specifies the output's line-by-line layout:

- Line 58 calls the **`Period` class's `getYears`, `getMonths` and `getDays` methods** to get the number of years, months and days between the birth date and time and the current date and time.
- Line 59 calls the **`Period` class's `toTotalMonths` method** to determine the total number of months in the `Period`.
- Line 60 calls the **`LocalDate` class's `until` method** to determine the difference between the `LocalDate birthday` and its first argument—the date part of the `now` returned by `now.toLocalDate()`. Method `until`'s second argument specifies the units in which the difference is returned. **`ChronoUnit.DAYS`** indicates the difference should be converted to days.
- Line 61 calls the **`Duration enum`'s `toHours`, `toMinutes` and `toSeconds` methods** to get the number of hours, minutes and seconds between the birth date and time and the current date and time.

Click here to view code image

```
48          // display age components
49          System.out.printf("""
50             You have been alive for %d years, %d months, and %d days
51
52                 Total months: %,d
53                   Total days: %,d
54                  Total hours: %,d
55                Total minutes: %,d
56                Total seconds: %,d
57             """,
58             period.getYears(), period.getMonths(), period.getDays(),
59             period.toTotalMonths(),
60             birthday.until(now.toLocalDate(), ChronoUnit.DAYS),
61             duration.toHours(), duration.toMinutes(), duration.toSeconds());
62       }
63
```

```
You have been alive for 54 years, 10 months, and 26 days

Total months:  658
Total days:    20,054
Total hours:   481,285
Total minutes: 28,877,144
Total seconds: 1,732,628,696
```

Method `displayDatesAndTimes` Displays the User's Birth Date and Time and the Current Date and Time Using Locale-Specific Formatting

Lines 69–70 create a `DateTimeFormatter` that's used in lines 74 and 76 to get string representations of the user's birth date and time and the current date and time for display:

Click here to view code image

```
64      // display birth date/time and current date/time in user's locale
65      public static void displayDatesAndTimes(
66         LocalDateTime birth, LocalDateTime current) {
67
68         // prepare format to display dates and times for user's locale
69         var formatter =
70            DateTimeFormatter.ofLocalizedDateTime(FormatStyle.MEDIUM);
71
72         // display formatted date and time
73         System.out.printf("%nBirth date/time: %s%n",
74            birth.format(formatter));
75         System.out.printf("Current date/time: %s%n%n",
76            current.format(formatter));
77      }
78   }
```

Line 70 calls the **`DateTimeFormatter` class's `static` `ofLocalizedDateTime` method** to get a `DateTimeFormatter` object that formats the date and time for the default locale of the user's system. The **`FormatStyle.MEDIUM`**[13] constant produces a date/time in the following format for our locale in the United States Eastern time zone:

```
Jul 20, 1969, 10:56:00 PM
```

13. "Enum Class `FormatStyle`," Accessed September 21, 2025. https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/time/format/FormatStyle.html.

Lines 74 and 76 convert the `LocalDateTime` objects to strings for display by calling the **`LocalDateTime` class's `format` method**. The argument is the `DateTimeFormatter` object (`formatter`, line 69) that specifies the formatting. The Date/Time API also provides extensive support for specifying custom locales, rather than simply working with the system's default locale.

5.12.2 Running the Program for a Different Locale Than Your Own

You can customize the Java runtime by passing **`-D` options** to the `java` command, as you'll see below. Each option is specified as `-D` flag followed immediately by a system property's name. For dates, times and numeric locale-specific formatting in other languages, countries and time zones, you can run the `AgeCalculator` program with the following options:

- **`-Duser.language`** specifies a language code.
- **`-Duser.country`** specifies a country code.
- **`-Duser.timezone`** specifies a timezone code.

For example, to set the language to Spanish, the country to Spain, and the time zone to Europe/Madrid, you can execute the program as follows:

Click here to view code image

```
java -Duser.language=es -Duser.country=ES
   -Duser.timezone=Europe/Madrid AgeCalculator
```

which produces an output similar to:

Click here to view code image

```
Enter your birthday (yyyy-MM-dd): 1969-07-21
Enter your birth time (HH:mm): 03:56

Birth date/time: 21 jul 1969, 3:56:00
Current date/time: 15 jun 2024, 17:42:39

You were born on a lunes
You have been alive for 54 years, 10 months, and 25 days

Total months:  658
Total days:    20.053
```

```
Total hours:   481.285
Total minutes: 28.877.146
Total seconds: 1.732.628.799
```

You can experiment with other languages, locales and time zones using the codes for each at the following links:

- ISO 639-1 Language Codes:

  Click here to view code image

  ```
  https://en.wikipedia.org/wiki/List_of_ISO_639_language_codes
  ```

- ISO 3166-1 Alpha-2 Country Codes:

  Click here to view code image

  ```
  https://en.wikipedia.org/wiki/ISO_3166-1_alpha-2
  ```

- Time Zone Codes:

  Click here to view code image

  ```
  https://en.wikipedia.org/wiki/List_of_tz_database_time_zones
  ```

Generative AI

1 Prompt genAIs with the Java code from this section and ask them to write the `java` commands for running the application in the language, country and time zone of each continent's largest city. Use the generated commands to test the program and obtain locale-specific outputs.

5.13 Wrap-Up

In this chapter, you created your first custom method declarations. You declared named constants using `enum` types. You used the class `RandomGenerator` to generate random numbers for simulations and game-playing. You learned that some applications require secure random numbers for which you can use class `SecureRandom`.

We introduced the `switch` enhancements and showed that they can more concisely represent `switch` logic. We also showed `switch` expressions that yield values—these simplify the code pattern in which a variable declaration is followed by a `switch` statement that assigns a value to the variable in every branch.

You learned about the scope of fields and local variables in a

class. We showed that multiple methods in a class can be overloaded by providing methods with the same name and different signatures—such methods typically perform the same or similar tasks using different types and/or numbers of parameters.

We used the `Math` class to introduce `static` class members, then discussed why `main` is `static`. You learned about some of the commonly used packages in the Java API. You saw how the method-call stack and stack frames keep track of the methods that have been called and where each method must return to when it completes its task. We also discussed Java's promotion rules for converting implicitly between primitive types and how to perform explicit conversions with cast operators.

You continued using generative AI to leverage your Java learning experience. In the Objects-Natural Case Study, you used capabilities of the Java Date/Time API to perform common date and time manipulations in an application that input your birth date and birth time, then determined how long you've been alive in years, months, days, hours, minutes and seconds, and on what day of the week you were born.

We'll discuss more details of methods in **Chapter 8, Real-World Modeling with Custom Classes**—where you'll declare your own classes, create objects of those classes and invoke their methods—and Chapter 22, Recursion and Big *O*.

In Chapter 6, you'll learn how to maintain lists and tables of data in arrays. You'll see a more elegant implementation of the program that rolls a die 60,000,000 times. You'll also learn how to access a program's command-line arguments that the JVM passes to method `main` when a program begins execution.

Special Section: Generative AI Exercises

For each of the following questions, try multiple genAIs to determine which performs best. Always verify the results. Keep in mind that they sometimes make mistakes and even hallucinate, creating unreliable information.

5.1 ***(Exercises for Practicing Methods)*** Prompt generative AIs to suggest introductory exercises with answers for practice creating `static` methods that you can call from Java's `main` method. Limit the suggestions to exercises that can be completed in a maximum of 50 lines of code.

5.2 ***(Prime Factorization)*** Prompt generative AIs to write a

program containing a method that determines the prime factors of a `BigInteger` and returns the prime factorization as an `ArrayList<BigInteger>`.

5.3 ***(Intro to Java Security)*** Prompt generative AIs to create a tutorial introducing Java's security features.

5.4 ***(IQ Test Sequence Predictor)*** Prompt generative AIs to generate a Java program that inputs an arbitrary sequence of integers, then predicts the next integer in the sequence.

5.5 ***(Date/Time API)*** Prompt generative AIs to write a tutorial on applications of Java's Date/Time API.

5.6 ***(Date/Time API)*** Prompt generative AIs to suggest Java Date/Time API exercises for practicing the features presented in Section 5.12's Objects-Natural Case Study on calculating your age.

number

7

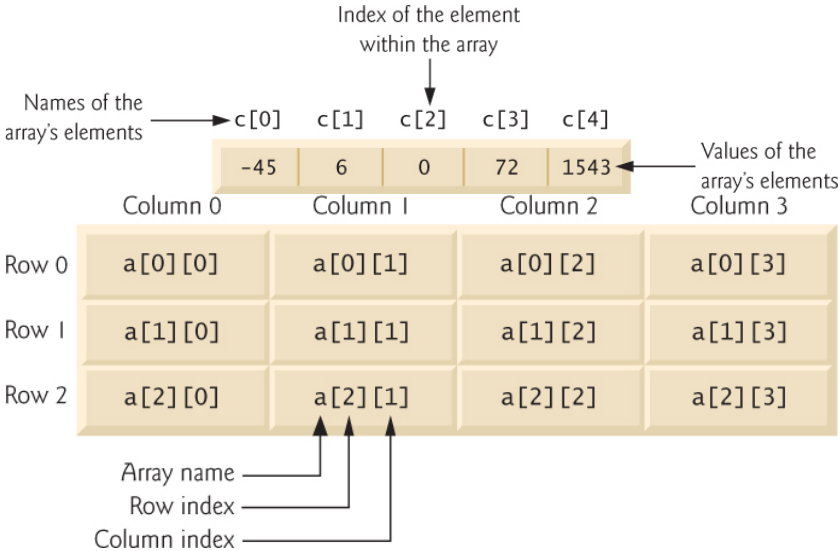
number contains the primitive int value 7

name

String object

"Paul"

name refers to a String object containing Paul



Arrays and `ArrayLists`

Objectives

In this chapter, you'll:

- Understand the differences between primitive and reference types.
- Use arrays to store data in and retrieve data from lists and tables of values.
- Declare arrays, initialize arrays and refer to individual

elements of arrays.

- Iterate through arrays with the enhanced `for` statement.
- Pass arrays to methods.
- Understand pass-by-value vs. pass-by-reference.
- Declare and manipulate multidimensional arrays.
- Define methods with variable-length argument lists.
- Read command-line arguments into a program.
- Perform predefined array manipulations using the methods of class `Arrays`.
- Use generative AI to leverage your Java learning experience.
- Continue our objects-natural approach with a case study on using the Java API's `ArrayList` class—a dynamically resizable array-like data structure with many capabilities beyond those of built-in Java arrays.

Outline

6.1 Introduction

6.2 Primitive Types vs. Reference Types

6.3 Arrays

6.4 Declaring and Creating Arrays

6.5 Creating and Initializing an Array

6.6 Array Initializers

6.7 Calculating Array Element Values

6.8 Totaling Array Elements

6.9 Intro to Visualization: Using a Bar Chart to Display Array Data Graphically

6.10 Using Array Elements as Counters

6.11 Analyzing Survey Results

6.12 Exception Handling

6.12.1 The `try` Statement

6.12.2 The `catch` Block

6.13 Enhanced `for` Statement: Totaling Array Elements

6.14 Passing Arrays to Methods

6.15 Pass-By-Value vs. Pass-By-Reference
6.16 Multidimensional Arrays
6.17 Variable-Length Argument Lists
6.18 Command-Line Arguments
6.19 Class `Arrays`
6.20 Objects-Natural Case Study: Intro to Collections and Class `ArrayList`
6.21 Wrap-Up
Special Section: Generative AI Exercises

6.1 Introduction

This chapter introduces **data structures**, which enable you to store **collections** of related data items. **Array** objects are data structures containing data items of the same type and make it convenient to process related groups of values. In this chapter, we:

- Examine the differences between primitive types and reference types and show that arrays can be used to store both.
- Discuss how to declare, create and initialize arrays, and demonstrate common array manipulations, such as calculating array-element values, totaling array elements, visualizing array data and using array elements as counters.
- Introduce Java's exception-handling mechanism, using it to process an error that occurs when a program attempts to access a non-existent array element.
- Demonstrate the enhanced `for` statement for processing an array's data more conveniently and with fewer potential errors than with the counter-controlled `for` statement.
- Introduce two-dimensional arrays for representing tables containing rows and columns of data.
- Declare and manipulate several two-dimensional arrays, iterating through them using nested counting `for` loops and nested enhanced `for` loops.
- Create methods that can be called with varying numbers of arguments and show how to manipulate the `main` method's

command-line arguments via its `args` parameter.

- Use array manipulation methods from class `Arrays` (package `java.util`) to avoid "reinventing the wheel."

You'll continue using generative AI to leverage your Java learning experience. This chapter's Objects-Natural Case Study introduces the `ArrayList` collection class—a reusable, reliable, powerful and efficient data structure that is similar to an array but is **dynamically resizable** to accommodate more or fewer elements.

6.2 Primitive Types vs. Reference Types

Java's types are either primitive types or **reference types**. The eight primitive types are `boolean`, `byte`, `char`, `short`, `int`, `long`, `float` and `double`—you've created variables of types `int` and `double`. You've also worked with objects of reference types including `Scanner`, `String`, `BigInteger`, `BigDecimal`, `RandomGenerator` and various types from the Java Date/Time API. **All nonprimitive types are reference types.**

Primitive-Type Variables Store Values

A primitive-type variable holds exactly one value of its declared type at a time. Consider the following `int` variable declaration:

```
int number = 7;
```

After this statement executes, `number` contains the value `7`, as in the following diagram:
When you assign a new value to `number`, the new value replaces `7`, and the previous value is lost.

Reference-Type Variables Store the Locations of Objects

Reference-type variables store **references** to objects that are located elsewhere in memory. For example, given the following declaration

```
String name = "Paul";
```

the variable `name` refers to a `String` object somewhere in memory containing `"Paul"`, as in the following diagram:
You use the variable `name` to refer to `String` object and call its methods. For example, the following statement calls the `String` object's `length` method and stores the result in the `int` primitive variable `length`:

```
int length = name.length();
```

You can invoke methods via reference-type variables, which refer to objects, but not via primitive-type variables.

6.3 Arrays

An array is a group of variables (called **elements**) containing data of the same type. Arrays are objects, so they're reference types—an array variable contains a reference to an array object in memory. The elements of an array can be primitive types or reference types (including other arrays, as shown in Section 6.16). To refer to a particular element in an array, we specify the name of the reference to the array and the **index** (that is, position number) of the element in the array.

The following diagram shows a visual representation of an `int` array named `c`:
Array names follow the same conventions as other variable names. This array contains five elements. A program refers to any one of these elements with an **array-access expression** that includes the name of the array followed by the index of the particular element in **square brackets**, `[ ]`—e.g., `c[0]`. The first element in every array has **index** `0`. Thus, the elements of the array `c` are `c[0]` (pronounced "`c` sub zero") through `c[4]`. The value of `c[0]` is `-45`, `c[2]` is `0` and `c[4]` is `1543`. An array-access expression can be used on the left side of an assignment to place a new value into an array element. For example, the following statement replaces `c[4]`'s value:

```
c[4] = 87;
```

An index must be a nonnegative integer expression. The highest index (in this case, 4) is always one less than the array's size (in this case, 5). Each array knows its own size (number of elements), which you can access via its **`length` instance variable**, as in

```
c.length
```

An array's length cannot be changed after it's created.

6.4 Declaring and Creating Arrays

An array object's elements occupy contiguous space in memory, enabling efficient access to the elements by their indices. To create an array object, you use an **array-creation expression** consisting of `new` followed by the array's element type and square brackets specifying the number of elements. Such an expression returns a reference that can be stored in an array variable. The following declaration and array-creation expression create an array object containing five `int` elements and store the array's reference in the array variable named `c`:

```
int[] c = new int[5];
```

When you create an array, each element receives a default value—zero for numeric primitive-type elements, `false` for `boolean` elements and `null` for reference-type elements. The value `null` indicates that a reference-type variable does not currently refer to an object. You'll soon see that you can provide non-default element values when you create an array.

Declaring an Array Variable and Creating an Array in Separate Statements

Creating an array can also be performed in two steps as follows:

Click here to view code image

```
int[] c; // declare the array variable
c = new int[5]; // create the array; assign to array variable
```

In the declaration, the square brackets following the type indicate that `c` is a variable that will refer to an array—the declaration does not reserve any memory for the array's elements. In the assignment statement, the array variable `c` receives the reference to a new array of five `int` elements.

Creating Multiple Arrays in One Statement

A program can create several arrays in a single declaration. The following declaration reserves 100 elements for `b` and 27 elements for `x`:

Click here to view code image

```
String[] b = new String[100], x = new String[27];
```

When the array's element type and the square brackets are combined at the beginning of the declaration, all the identifiers in the declaration are array variables. In this case, variables `b` and `x` refer to `String` arrays. For readability and clarity, declare only one variable per declaration. The preceding declaration is equivalent to:

Click here to view code image

```
String[] b = new String[100]; // create array b
String[] x = new String[27]; // create array x
```

Placement of the Square Brackets in Array Variable Declarations

Declaring multiple array variables in a single declaration can lead to subtle errors. Consider the declaration

```
int[] a, b, c;
```

This declares `a`, `b` and `c` as array variables. Again, placing square brackets directly following the type indicates that all the identifiers in the declaration are array variables. However, if only `a` is intended to be an array variable, and `b` and `c` are intended to be individual `int` variables, then the declaration should be

```
int a[], b, c;
```

In this declaration, `int` distributes to each variable, but the square brackets apply only to the variable `a`. So, `a` can store a reference to an array of `int`s, and `b` and `c` can store individual `int` values. Declaring each variable in a separate variable-declaration statement eliminates this confusion.

Primitive-Type vs. Reference-Type Elements

A program can declare arrays with elements of any type:

- Every element of a primitive-type array is a variable that contains a value of the array's declared element type. For example, every `int` array element is a variable that stores an `int` value.
- Every element of a reference-type array is a variable that can refer to an object of the array's declared element type. For example, every `String` array element is a variable that can refer to a `String` object.

6.5 Creating and Initializing an Array

The program of Fig. 6.1 uses the keyword `new` to create an array of five `int` elements, which are initially zero—the default initial value for `int` array elements. Line 7 declares `array`—a reference-type variable capable of referring to an array of `int` elements—then initializes it with a reference to an array object containing five `int` elements. Line 9 outputs the column headings. The first column contains each array element's index (0–4), and the second column contains each array element's default initial value (0).

Click here to view code image

```
1  // Fig. 6.1: InitArray.java
2  // Initializing the elements of an array to default values of zero.
3
4  public class InitArray {
5     public static void main(String[] args) {
6        // declare variable array and initialize it with an array object
7        int[] array = new int[5]; // create the array object
8
9        System.out.printf("%s%8s%n", "Index", "Value"); // column headings
10
11       // output each array element's value
12       for (int counter = 0; counter < array.length; ++counter) {
13          System.out.printf("%5d%8d%n", counter, array[counter]);
14       }
15    }
16 }
```

```
Index    Value
    0        0
    1        0
    2        0
    3        0
    4        0
```

Fig. 6.1 | Initializing the elements of an array to default values of zero.

The `for` statement (lines 12–14) outputs the index (represented by `counter`) and value of each array element (represented by `array[counter]`). The control variable `counter` is initially `0` because index values start at 0. Using **zero-based counting** allows the loop to access every element. The `for`'s loop-continuation condition uses the expression `array.length` (line 12) to determine the array's length. In this example, the length is five, so the loop continues executing as long as `counter`'s value is less than 5. A five-element array's highest index value is 4, so using the less-than operator in the loop-continuation condition ensures the loop does not attempt to access an element beyond the end of the array—during the loop's final iteration, `counter` is `4`. We'll soon see what Java does when encountering an out-of-range index at execution time.

Generative AI

1 Prompt genAIs to find and correct the error(s) in the following code:

Click here to view code image

```
int[] values = new int[10];

for (int i = 0; i <= values.length; ++i) {
   System.out.println(values[i]);
}
```

Confirm that their responses indicate the loop-continuation test causes the code to reference an out-of-bounds array element (`values[10]`) and change the `<=` operator to `<`.

6.6 Array Initializers

You can create an array and initialize its elements with an **array initializer**—a comma-separated list of expressions enclosed in braces. In this case, the array length is determined by the number of elements in the array initializer. For example,

Click here to view code image

```
int[] n = {10, 20, 30, 40, 50};
```

creates a five-element array with index values `0`–`4`. Element `n[0]` is initialized to `10`, `n[1]` to `20`, etc. When the compiler encounters an array declaration that includes an array initializer, it counts the number of initializers in the list to determine the array's size and creates a new array of that size for you.

Figure 6.2 initializes an integer array with five values (line 7) and displays the array in tabular format. The code for displaying the array elements (lines 12–14) is identical to that in Fig. 6.1 (lines 12–14).

Click here to view code image

```
1  // Fig. 6.2: InitArray.java
2  // Initializing the elements of an array with an array initializer.
3
4  public class InitArray {
5     public static void main(String[] args) {
6        // array initializer specifies the initial value for each element
7        int[] array = {32, 27, 64, 18, 95};
8
9        System.out.printf("%s%8s%n", "Index", "Value"); // column headings
10
11       // output each array element's value
12       for (int counter = 0; counter < array.length; ++counter) {
13          System.out.printf("%5d%8d%n", counter, array[counter]);
14       }
15    }
16 }
```

```
Index    Value
    0       32
    1       27
    2       64
    3       18
    4       95
```

Fig. 6.2 | Initializing the elements of an array with an array initializer.

Rather than using array initializers, as in line 7, array element values are often read into a program. For example, you could input the values from a user or a file on disk, as discussed in Chapter 11, Files, I/O Streams, JSON Serialization & CSV Files. Reading the data into a program (rather than "hand coding" it into the program) makes the program more reusable because it can be used with different data sets.

Generative AI

1 Prompt genAIs to write a `for` loop that copies the elements of the `int` array `values` into the `int` array `otherValues`, which has the same number of elements as `values`. Compare their responses to the following code:

Click here to view code image

```
for (int counter = 0; counter < values.length; ++counter) {
   otherValues[counter] = values[counter];
}
```

6.7 Calculating Array Element Values

Figure 6.3 creates a five-element array, assigns an even integer from 2 through 10 to each element, then displays the array in tabular format. The `for` statement in lines 10–12 calculates an array element's value by multiplying the current value of the control variable `counter` by `2`, then adding `2`. Lines 17–19 display each array element's index and value.

Click here to view code image

```
1  // Fig. 6.3: InitArray.java
2  // Calculating the values to be placed into the elements of an array.
3
4  public class InitArray {
5     public static void main(String[] args) {
6        final int ARRAY_LENGTH = 5; // declare constant
7        int[] array = new int[ARRAY_LENGTH]; // create array
8
9        // calculate value for each array element
10       for (int counter = 0; counter < array.length; ++counter) {
11          array[counter] = 2 + 2 * counter;
12       }
13
14       System.out.printf("%s%8s%n", "Index", "Value"); // column headings
15
16       // output each array element's value
17       for (int counter = 0; counter < array.length; ++counter) {
18          System.out.printf("%5d%8d%n", counter, array[counter]);
19       }
20    }
21 }
```

```
Index    Value
    0        2
    1        4
    2        6
    3        8
    4       10
```

Fig. 6.3 | Calculating the values to be placed into the elements of an array.

We specified the array's size using the `ARRAY_LENGTH` constant variable (line 6) declared with the `final` modifier and initialized to `5`. Constant variables must be initialized before use and cannot be modified thereafter. Attempting to modify a `final` variable after it's initialized causes a compilation error. Like `enum` constants, constant variables are named constants. They often make programs more readable. A named constant such as `ARRAY_LENGTH` clearly indicates its purpose, whereas a literal value such as `5` could have different meanings based on its context. Constant variables use all uppercase letters by convention, and multiword names should separate each word from the next with an underscore (_) as in `ARRAY_LENGTH`.

Generative AI

1 Prompt genAIs to rewrite the program of Fig. 6.3 to calculate each element's value as the square of that element's index.

6.8 Totaling Array Elements

Array elements often stored values for use in calculations. For example, if an array's elements represent exam grades, a professor may wish to total them and use the sum to calculate the class average. Lines 10–12 in Fig. 6.4 sum the values in the 10-element `int` array `grades` (created and initialized at line 6). In Section 6.13, we'll reimplement this example with the "enhanced `for`" statement, which does not require a counter.

Click here to view code image

```
1  // Fig. 6.4: SumArray.java
2  // Computing the sum of the elements of an array.
3
4  public class SumArray {
5     public static void main(String[] args) {
```

```
6          int[] grades = {87, 68, 94, 100, 83, 78, 85, 91, 76, 87};
7          int total = 0;
8
9          // add each element's value to total
10         for (int counter = 0; counter < grades.length; ++counter) {
11            total += grades[counter];
12         }
13
14         System.out.printf("Total of array elements: %d%n", total);
15      }
16   }
```

```
Total of array elements: 849
```

Fig. 6.4 | Computing the sum of the elements of an array.

Generative AI

1 Prompt genAIs to write Java code that calculates the product of the elements in the array `values` containing the integers 1–100. Ask them to explain the steps they used to create the code in their responses.

6.9 Intro to Visualization: Using a Bar Chart to Display Array Data Graphically

Many programs present data to users graphically—a process called **visualization**. For example, numeric values are often displayed as bars in a bar chart, with longer bars represent proportionally larger numeric values. One way to display numeric data graphically is with a bar chart that shows each numeric value as a bar of asterisks (*).

Professors often like to review an exam's grade distribution. They might visualize this with a graph showing the number of grades in several categories. Suppose the grades were 87, 68, 94, 100, 83, 78, 85, 91, 76 and 87. They include one grade of 100, two grades in the 90s, four in the 80s, two in the 70s, one in the 60s and none below 60. Figure 6.5 stores this grade distribution data in an 11-element array. Each element represents a grade category. For

example, `distribution[0]` stores the number of grades in the range 0–9, `distribution[7]` the number of grades in the range 70–79 and `distribution[10]` the number of 100 grades.

Click here to view code image

```
1  // Fig. 6.5: BarChart.java
2  // Bar chart printing program.
3
4  public class BarChart {
5     public static void main(String[] args) {
6        int[] distribution = {0, 0, 0, 0, 0, 0, 1, 2, 4, 2, 1};
7
8        System.out.println("Grade distribution:");
9
10       // for each distribution element, output a bar of the chart
11       for (int counter = 0; counter < distribution.length; ++counter) {
12          // output bar label ("00-09: ", ..., "90-99: ", "100: ")
13          if (counter == 10) {
14             System.out.printf("%5d: ", 100);
15          }
16          else {
17             System.out.printf("%02d-%02d: ",
18                counter * 10, counter * 10 + 9);
19          }
20
21          // print bar of asterisks
22          for (int stars = 0; stars < distribution[counter]; ++stars) {
23             System.out.print("*");
24          }
25
26          System.out.println();
27       }
28    }
29 }
```

```
Grade distribution:
00-09:
10-19:
20-29:
30-39:
40-49:
50-59:
60-69: *
70-79: **
80-89: ****
90-99: **
  100: *
```

Fig. 6.5 | Bar chart printing program.

The program graphs the array's information as a bar chart with each grade range followed by a bar of asterisks indicating the number of grades in that range. To label each bar, lines 13–19 output a grade range (e.g., `"70-79: "`) based on the current `counter` value. When `counter` is `10`, line 14 outputs `100` with a field width of 5, followed by a colon and a space, to align the label `"100: "` with the other bar labels. In this example, no students received a grade below 60, so `distribution[0]` through `distribution[5]` each contain `0`, and no asterisks are displayed next to the first six grade ranges. In line 17, the format specifier `%02d` indicates that an `int` value should be formatted as a field of two digits. The **`0` flag** in the format specifier displays a leading `0` for values with fewer digits than the field width (`2`). Appendix B, Formatted Output, discuss format specifiers and format flags.

Generating the Bars

The nested `for` statement (lines 22–24) outputs the bars—the outer loop iterates over the grade ranges, and the nested loop prints the asterisks for each range. Note the loop-continuation condition at line 22 (`stars < distribution[counter]`). When the program reaches the nested `for`, the loop counts from `0` to one less than `distribution[counter]`, thus using a value in `distribution` to determine the number of asterisks to display.

Lines 22–26 in Fig. 6.5 could be replaced with

Click here to view code image

```
System.out.println("*".repeat(distribution[counter]));
```

Recall that a string literal like `"*"` is a `String` object. The preceding statement calls the **`String` method `repeat`** on the `String` object `"*"` to repeat it the number of times specified by `repeat`'s argument. The method returns a new `String` containing the repeated text.

Generative AI

1 Prompt genAIs for a list of flags used in Java `System.out.printf` format strings.

6.10 Using Array Elements as Counters

Sometimes, programs use counters to summarize data, such as survey results. In Fig. 5.3, we used separate counters in our die-rolling program to track the frequencies if each face on a six-sided die as the program rolled the die 60,000,000 times. Figure 6.6 presents an array version of Fig. 5.3.

Click here to view code image

```
1  // Fig. 6.6: RollDie.java
2  // Die-rolling program using arrays instead of switch.
3  import java.util.random.RandomGenerator;
4
5  public class RollDie {
6     public static void main(String[] args) {
7        var randomNumbers = RandomGenerator.getDefault();
8        int[] frequency = new int[7]; // array of frequency counters
9
10       // roll die 60,000,000 times; use die value as frequency index
11       for (int roll = 1; roll <= 60_000_000; ++roll) {
12          ++frequency[randomNumbers.nextInt(1, 7)];
13       }
14
15       System.out.printf("%s%12s%n", "Face", "Frequency");
16
17       // output each array element's value
18       for (int face = 1; face < frequency.length; ++face) {
19          System.out.printf("%4d%12d%n", face, frequency[face]);
20       }
21    }
22 }
```

```
Face   Frequency
   1    10001418
   2     9999022
   3     9996511
   4    10001187
   5    10002132
   6     9999730
```

Fig. 6.6 | Die-rolling program using arrays instead of `switch`.

Figure 6.6 uses the array `frequency` (line 8) to count the occurrences of each face of the die. Line 12 of this program replaces lines 19–29 of Fig. 5.3. Line 12 uses the random value to determine which `frequency` element to increment. The calculation in line 12

produces random numbers from 1 to 6, so the array `frequency` must be large enough to store six counters. We chose to use a seven-element array in which we ignore `frequency[0]` because it's clearer to have the face value 1 increment the counter variable `frequency[1]` than `frequency[0]`. Using a seven-element array with indices 0–6 ensures that it has an index for each possible die face (1–6). In line 12, the calculation inside the square brackets evaluates first to determine the array element to increment, then the `++` operator adds one to that element. We also replaced lines 33–35 from Fig. 5.3 by looping through array `frequency` to output the results (lines 18–20). In Chapter 14, we'll replace lines 11–13 and 18–20 with a single statement using functional programming capabilities!

Generative AI

1 Prompt genAIs to write a program that uses an array to summarize 100 flips of a coin and displays the number of heads and tails. Compare the generated code to the following code. If the generated code is significantly different, ask them to explain their code and the steps used to develop it.

Click here to view code image

```
1  // CoinFlipper.java
2  // Summarizing coin flips.
3  import java.util.random.RandomGenerator;
4
5  public class CoinFlipper {
6     public static void main(String[] args) {
7        RandomGenerator randomNumbers = RandomGenerator.getDefault();
8
9        // constants representing heads and tails
10       final int HEADS = 0;
11       final int TAILS = 1;
12
13       // frequency[0] for heads, frequencies[1] for tails
14       int[] frequency = new int[2];
15
16       // simulate 100 coin flips
17       for (int counter = 0; counter < 100; ++counter) {
18          ++frequency[randomNumbers.nextInt(2)];
19       }
20
21       System.out.printf("Number of heads: %d%n", frequency[HEADS]);
22       System.out.printf("Number of tails: %d%n", frequency[TAILS]);
```

```
23       }
24    }
```

```
Number of heads: 46
Number of tails: 54
```

6.11 Analyzing Survey Results

Next, let's use arrays to summarize data collected in a survey. Consider the following problem statement:

> Twenty students were asked to rate the student cafeteria's food quality on a scale of 1 to 5, with 1 being "awful" and 5 being "excellent." Place the responses in an integer array and determine the frequency of each rating.

This is a typical use of arrays. We wish to summarize the responses of each type (that is, 1–5). Array `responses` (Fig. 6.7, lines 7–8) is a 20-element integer array containing the students' survey responses. The array's last value is intentionally an incorrect response (`14`)—we'll use this value to show that Java validates array index values at runtime.

Click here to view code image

```
1  // Fig. 6.7: StudentPoll.java
2  // Poll analysis program.
3
4  public class StudentPoll {
5     public static void main(String[] args) {
6        // student response array (more typically, input at runtime)
7        int[] responses =
8           {1, 2, 5, 4, 3, 5, 2, 1, 3, 3, 1, 4, 3, 3, 3, 2, 3, 3, 2, 14};
9        int[] frequency = new int[6]; // array of frequency counters
10
11       // for each answer, select responses element and use that value
12       // as frequency index to determine array element to increment
13       for (int answer = 0; answer < responses.length; ++answer) {
14          try {
15             ++frequency[responses[answer]];
16          }
```

```
            catch (ArrayIndexOutOfBoundsException e) {
               System.out.println(e); // invokes toString method
               System.out.printf("   responses[%d] = %d%n%n",
                  answer, responses[answer]);
            }
         }

         System.out.printf("%s%10s%n", "Rating", "Frequency");

         // output each array element's value
         for (int rating = 1; rating < frequency.length; ++rating) {
            System.out.printf("%6d%10d%n", rating, frequency[rating]);
         }
      }
   }
```

```
java.lang.ArrayIndexOutOfBoundsException: Index 14 out of bounds for length 6
   responses[19] = 14

Rating Frequency
     1         3
     2         4
     3         8
     4         2
     5         2
```

Fig. 6.7 | Poll analysis program.

The `frequency` Array

We use the six-element array `frequency` (line 9) to count the number of occurrences of each response. Each element (except element 0) is used as a counter for one of the possible survey-response values—`frequency[1]` counts the number of students who rated the food as 1, `frequency[2]` counts the number of students who rated the food as 2, and so on.

Summarizing the Results

The `for` statement (lines 13–22) reads the responses from the array `responses` one at a time and increments one of the counters `frequency[1]` to `frequency[5]`; we ignore `frequency[0]` because the survey responses are limited to the range 1–5. The key statement in the loop is line 15. This statement increments the appropriate `frequency` counter as determined by the value of `responses[answer]`.

Let's step through the first few iterations of the `for` statement:

- When `answer` is `0`, `responses[answer]` is the value of `responses[0]` (that is, `1`—line 8). So, `frequency[responses[answer]]` evaluates to `frequency[1]`, and counter `frequency[1]` is incremented by one. To evaluate the expression, we begin with the value in the innermost set of brackets (`answer`, currently `0`). The value of `answer` is plugged into the expression, and the next set of brackets (`responses[answer]`) is evaluated. That value is used as the `frequency` array index to determine which counter to increment (in this case, `frequency[1]`).
- The next time through the loop `answer` is `1`, and `responses[answer]` is the value of `responses[1]` (that is, `2`—line 8). So, `frequency[responses[answer]]` is interpreted as `frequency[2]`, causing `frequency[2]` to be incremented.
- When `answer` is `2`, `responses[answer]` is the value of `responses[2]` (that is, `5`—line 8). So, `frequency[responses[answer]]` is interpreted as `frequency[5]`, causing `frequency[5]` to be incremented, and so on.

Regardless of the number of responses processed, only a six-element array (in which we ignore element zero) is required to summarize the results because all the correct responses are from 1 to 5, and the index values for a six-element array are 0–5.

Processing the Incorrect Response

In the program's output, the `Frequency` column summarizes only 19 of the 20 values in the `responses` array. The `responses` array's last element contains an (intentionally) incorrect response that was not counted. All indices must be greater than or equal to 0 and less than the array's length. The process of validating indices at runtime is called **array bounds checking**. Any attempt to access an element outside that range results in a runtime error known as an **`ArrayIndexOutOfBoundsException`**. The output's first line shows the message associated with that error. The message includes the exception's package and class name, and an error message indicating the index value that was out of bounds and the length of the array. The next section introduces Java's exception-handling mechanism used in lines 14–21 to display the first two lines of this program's output when the `Array-IndexOutOfBoundsException`

occurred. Exception handling enables programs to handle unexpected values gracefully without crashing.

6.12 Exception Handling

An **exception** indicates a problem that occurs while a program executes. **Exception handling** helps you create **fault-tolerant programs** that can resolve (or handle) exceptions. Sometimes, this allows a program to continue executing as if no problems were encountered. For example, the `StudentPoll` program still displays results (Fig. 6.7), even though one of the responses was out of range. More severe problems might prevent a program from continuing normal execution, instead requiring it to notify the user of the problem, then terminate. When the JVM or a method detects a problem, such as an invalid array index or an invalid method argument, it **throws an exception**—that is, an exception occurs. Methods in your own classes can also throw exceptions, as you'll see in Chapter 8.

6.12.1 The `try` Statement

To handle an exception, use a **`try` statement** (lines 14–21 of Fig. 6.7). The **`try` block** (lines 14–16) contains the code that might throw an exception, and the **`catch` block** (lines 17–21) contains the code that handles the exception if one occurs. You can have many `catch`es to handle various exceptions that might be thrown in the corresponding `try` block. When line 15 correctly increments a `frequency` array element, lines 17–21 are skipped. The braces that delimit the `try` and `catch` blocks are required.

6.12.2 The `catch` Block

When the program encounters the invalid value `14` in the `responses` array, it attempts to add `1` to `frequency[14]`, which is outside the array's bounds—the `frequency` array has only six elements (with indexes 0–5). The JVM performs array bounds checking at execution time. When line 15 attempts to access an out-of-bounds element, the JVM throws an `ArrayIndexOutOfBoundsException` to notify the program of this problem. At this point, the `try` block terminates, and the `catch` block begins executing. If you declared local variables in the `try` block, they're now out of scope (and no longer exist), so they're not

accessible in the `catch` block. The `catch` block declares an `ArrayIndexOutOfBoundsException` exception parameter (`e`) to handle exceptions of that type. You can use the parameter's identifier to interact with a caught exception object inside the `catch` block.

To prevent `ArrayIndexOutOfBoundsException`s when writing code that accesses array elements, ensure that each array index is greater than or equal to 0 and less than the array's length. For example, the code in Fig. 6.7 could have performed data validation on the values in `responses` to ensure they were valid response values before using them as array indices for the `frequency` array.

Systems in industry that have undergone extensive testing are still likely to contain bugs. Our preference for industrial-strength systems is to catch and deal with runtime exceptions, such as `ArrayIndexOutOfBoundsException`s, to ensure that a system either stays up and running or degrades gracefully and to inform the system's developers of the problem.

`toString` Method of the Exception Parameter

When lines 17–21 of Fig. 6.7 catch the exception, the program displays a message indicating the problem that occurred. Every object has a `String` representation. When you display an object using `System.out`'s `print`, `println` and `printf` methods, they implicitly call the object's `toString` method to get its `String` representation.[1] Line 18 implicitly calls the exception object's `toString` method to get the exception object's error message `String` and display it. Lines 19–20 show the `responses[answer]` value that caused the problem. Once the message is displayed in this example, the exception is considered handled, and the program continues with the statement after the `catch` block's closing brace. In this example, the end of the `for` statement is reached (line 22), so the program continues with the increment of the control variable in line 13. We discuss exception handling again in Chapter 8 and more deeply in Chapter 10.

1. Section 8.6 discusses `toString` in more detail.

Generative AI

1 Prompt genAIs with the code in Fig. 6.7. Ask them to rewrite the code to remove the `try` statement but still prevent `ArrayIndexOutOfBoundsException`s. Ask them to explain the updated code. Execute the generated code to confirm that it runs correctly

and produces the same two-column summary of the response frequencies.

6.13 Enhanced `for` Statement: Totaling Array Elements

Figure 6.8 reimplements Fig. 6.4, which totaled the integers in an `int` array named `grades`. Lines 10–12 introduce the **enhanced `for` statement** for iterating through an array's elements without using a counter. This eliminates several error possibilities, such as improperly specifying the control variable's initial value, the loop-continuation test and the increment expression, thus avoiding the possibility of "stepping outside" the array.

Click here to view code image

```
1  // Fig. 6.8: EnhancedForTest.java
2  // Using the enhanced for statement to total integers in an array.
3
4  public class EnhancedForTest {
5     public static void main(String[] args) {
6        int[] grades = {87, 68, 94, 100, 83, 78, 85, 91, 76, 87};
7        int total = 0;
8
9        // add each element's value to total
10       for (int grade : grades) {
11          total += grade;
12       }
13
14       System.out.printf("Total of array elements: %d%n", total);
15    }
16 }
```

```
Total of array elements: 849
```

Fig. 6.8 | Using the enhanced `for` statement to total integers in an array.

The enhanced `for` statement's header (line 10) declares the variable `grade` as an `int`, because the `grades` array contains `int` values. The variable declaration and the array are separated by a required colon (`:`). The loop selects one `int` value from `grades`

during each iteration and assigns it to `grade`, which the program processes in the loop's body. This enhanced `for` statement's header can be read as "for each `grade` in `grades`." Line 11 adds the `grade` variable's current value to the `total`. The loop automatically terminates after it processes the `grades` array's last element.

Equivalent Counting `for` Loop

Lines 10–12 are equivalent to the following counter-controlled iteration used in lines 10–12 of Fig. 6.4 to total the integers in `grades`:

Click here to view code image

```
for (int counter = 0; counter < grades.length; ++counter) {
   total += grades[counter];
}
```

The enhanced `for` statement hides the counting details from you.

When to Use the Enhanced `for` Statement

The enhanced `for` statement can be used to read array elements' values,[2] but not modify them.[3] If your program needs to modify elements, use a counter-controlled `for` statement. You should use the enhanced `for` statement when code looping through an array does not need access to each array element's index. For example, totaling the integers in an array requires only the element values—each element's index is irrelevant.

2. Section 6.20 will show that the enhanced `for` statement can also be used with other collections of data.
3. Modifying an element of a primitive-type array means changing the element's value. Modifying an element of a reference-type array means making the element to refer to a different object.

Functional Programming

The `for` statement and the enhanced `for` statement each iterate sequentially from a starting value to an ending value. In Chapter 14, Functional Programming with Lambdas & Streams, you'll see an elegant, more concise and less error-prone means for iterating through arrays and other collections in a manner that enables some iterations to occur in parallel with others to achieve better multi-core system performance.

Generative AI

1 Prompt genAIs to rewrite the code in Fig. 6.2 to use an enhanced `for` loop and produce the same output. Compare the generated code's enhanced `for` loop to Fig. 6.2's counter-controlled `for` loop to examine how the generated code determines and displays the array elements' index values.

6.14 Passing Arrays to Methods

This section demonstrates how to pass arrays and individual array elements as arguments to methods. To pass an array argument to a method, use the array name without brackets. For example, if `hourlyTemperatures` is declared as

Click here to view code image

```
double[] hourlyTemperatures = new double[24];
```

then the method call

Click here to view code image

```
processTemps(hourlyTemperatures);
```

passes the reference of array `hourlyTemperatures` to method `processTemps`. Recall that every array object "knows" its own length via its `length` instance variable, so when we pass an array object's reference into a method, we need not pass the array's length as an additional argument.

Array Parameters

For a method to receive an array reference through a method call, its parameter list must specify an array parameter. For example, the method header for method `processTemps` might be written as

Click here to view code image

```
void processTemps(double[] temps)
```

indicating that parameter `temps` receives the reference of a `double` array. The method call passes array `hourlyTemperature`'s reference,

so when the called method uses the array variable `temps`, it refers to the same array object as `hourlyTemperatures` in the caller.

When an argument to a method is an entire array or an individual array element of a reference type, the called method receives a copy of the reference. However, when an argument to a method is an individual array element of a primitive type, the called method receives a copy of the element's value. Such primitive values are called **scalars** or **scalar quantities**. To pass an individual array element to a method, use the indexed name of the array element as an argument in the method call (as you'll see momentarily).

Passing Arrays and Individual Array Elements to Methods

Figure 6.9 demonstrates passing an entire array and passing a primitive-type array element to a method. The `main` method calls `static` methods `modifyArray` and `modifyElement` at lines 18 and 30, respectively. Recall that a `static` method can invoke other `static` methods of the same class without using the class name and a dot (`.`) separator.

Click here to view code image

```
1  // Fig. 6.9: PassArray.java
2  // Passing arrays and individual array elements to methods.
3
4  public class PassArray {
5     // main creates array and calls modifyArray and modifyElement
6     public static void main(String[] args) {
7        int[] array = {1, 2, 3, 4, 5};
8
9        System.out.printf(
10          "Effects of passing reference to entire array:%n" +
11          "The values of the original array are:%n");
12
13       // output original array elements
14       for (int value : array) {
15          System.out.printf("   %d", value);
16       }
17
18       modifyArray(array); // pass array reference
19       System.out.printf("%n%nThe values of the modified array are:%n");
20
21       // output modified array elements
22       for (int value : array) {
23          System.out.printf("   %d", value);
```

```
24          }
25
26          System.out.printf(
27             "%n%nEffects of passing array element value:%n" +
28             "array[3] before calling modifyElement: %d%n", array[3]);
29
30          modifyElement(array[3]); // attempt to modify array[3]
31          System.out.printf(
32             "array[3] after modifyElement completes: %d%n", array[3]);
33       }
34
35       // multiply each element of an array by 2
36       public static void modifyArray(int[] array2) {
37          for (int counter = 0; counter < array2.length; ++counter) {
38             array2[counter] *= 2;
39          }
40       }
41
42       // multiply argument by 2
43       public static void modifyElement(int element) {
44          element *= 2;
45          System.out.printf(
46             "Value of element in modifyElement: %d%n", element);
47       }
48    }
```

```
Effects of passing reference to entire array:
The values of the original array are:
   1   2   3   4   5

The values of the modified array are:
   2   4   6   8   10

Effects of passing array element value:
array[3] before calling modifyElement: 8
Value of element in modifyElement: 16
array[3] after modifyElement completes: 8
```

Fig. 6.9 | Passing arrays and individual array elements to methods.

The enhanced `for` statement in lines 14–16 outputs `array`'s elements. Line 18 invokes `modifyArray` (lines 36–40), passing `array` as an argument. The method receives a copy of `array`'s reference and uses it to multiply each of `array`'s elements by 2. To show that `array`'s elements were modified, lines 22–24 output the elements again. We could not use the enhanced `for` statement in lines 37–39 because we're modifying `array`'s values.

Next, we demonstrate that when a copy of a primitive-type array

element is passed to a method, modifying the copy does not affect the original value of that element in the calling method's array. Lines 26–28 output `array[3]`'s value before invoking `modifyElement`. Remember that the value of this element is now 8 after it was modified during the call to `modifyArray`. Line 30 calls `modifyElement` and passes `array[3]` as an argument. This is just one `int` value (8) in `array`, so the program passes a copy of `array[3]`'s value. Method `modifyElement` (lines 43–47) multiplies its argument by 2, stores the result in its parameter `element`, then outputs `element`'s value (16). Since method parameters, like local variables, cease to exist when the method in which they're declared completes execution, the method parameter `element` is destroyed when `modifyElement` terminates. When the program returns control to `main`, lines 31–32 output the unmodified `array[3]` value (i.e., 8).

Generative AI

1 Prompt genAIs to create a Java method named `sum` that `main` can call to calculate and return the sum of an `int` array's elements, then test the method's code in a program. Execute the generated code to ensure that it works.

6.15 Pass-By-Value vs. Pass-By-Reference

The preceding example demonstrated passing arrays and primitive-type array elements as arguments to methods. Let's take a closer look at the two ways to pass arguments to methods in many programming languages—**pass-by-value** and **pass-by-reference**.

When an argument is passed by value, a copy of the argument's value is passed to the called method. The called method works exclusively with the copy. Changes to the called method's copy do not affect the original variable's value in the caller.

When an argument is passed by reference, the called method can access the argument's value in the caller directly and modify that data, if necessary. Pass-by-reference improves performance by eliminating the need to copy possibly large amounts of data.

Unlike some other languages, Java passes all arguments to methods by value, meaning it passes a copy of each argument. A method call can pass two types of values to a method—copies of primitive values (e.g., values of type `int` and `double`) and copies of references to objects. Objects themselves cannot be passed to

methods.

When a method modifies a primitive-type parameter, changes to the parameter have no effect on the original argument value in the calling method. For example, in Fig. 6.9, when line 30 in `main` passed `array[3]` to method `modifyElement`, the statement in line 44 that multiplied the parameter `element` by `2` did not affect `array[3]`'s value in `main`. This is also true for reference-type parameters. If you modify a reference-type parameter so it refers to another object, only the method's parameter refers to the new object—the reference stored in the caller's variable still refers to the original object.

When passing reference-type variables, the method can modify the referenced object by calling its methods but cannot change the reference itself. Since the reference stored in the parameter is a copy of the reference passed as an argument, the parameter in the called method and the argument in the calling method refer to the same object in memory. For example, in Fig. 6.9, parameter `array2` in method `modifyArray` and variable `array` in `main` refer to the same array object in memory. Any changes made using the parameter `array2` are carried out on the object that `array` references in the calling method. In Fig. 6.9, the changes made in `modifyArray` using `array2` affect the contents of the array object referenced by `array` in `main`. Thus, the called method can use the reference it receives to manipulate the caller's object directly.

Passing references to arrays instead of the array objects themselves makes sense for performance reasons. Because arguments are passed by value, if array objects were passed, a copy of each element would be passed. For large arrays, this would waste time and consume considerable storage for the element copies.

6.16 Multidimensional Arrays

Multidimensional arrays with two dimensions often represent tables of values with data arranged in rows and columns. To identify a particular table element, you specify two indices. By convention, the first identifies the element's row and the second its column. Arrays that require two indices to identify each element are called **two-dimensional arrays**. Multidimensional arrays can have two or more dimensions. Java's multidimensional arrays are actually one-

dimensional arrays whose elements refer to one-dimensional arrays. The following diagram illustrates a two-dimensional array named `a` with three rows and four columns (i.e., a three-by-four array). An array with *m* rows and *n* columns is called an ***m*-by-*n* array**.

Every element in array `a` is identified by an array-access expression of the form `a[`*row*`][`*column*`]` in which `a` is the array's name, and *row* and *column* are indices that uniquely identify each element by row index and column index. The element names in row 0 all have a first index of `0`, and the element names in column 3 all have a second index of `3`.

Arrays of One-Dimensional Arrays

Like one-dimensional arrays, multidimensional arrays can be initialized with array initializers in declarations. A two-dimensional array `b` with two rows and two columns could be declared and initialized with **nested array initializers** as follows:

```
int[][] b = {{1, 2}, {3, 4}};
```

The initial values are grouped by row in braces. So `1` and `2` initialize `b[0][0]` and `b[0][1]`, respectively, and `3` and `4` initialize `b[1][0]` and `b[1][1]`, respectively. The compiler counts the number of nested array initializers (represented by sets of braces within the outer braces) to determine the number of rows in array `b`. The compiler counts the initializer values in a row's nested array initializer to determine its number of columns. As we'll see, this means that rows can have different lengths, enabling "jagged arrays."

Multidimensional arrays are maintained as arrays of one-dimensional arrays. Therefore, array `b`'s rows in the preceding declaration are one-dimensional arrays—one containing the values in the first nested array initializer `{1, 2}` and one containing the values in the second nested array initializer `{3, 4}`. So, array `b` is a one-dimensional array of two elements, each of which refers to a one-dimensional array of two `int` values.

Two-Dimensional Arrays with Rows of Different Lengths

The manner in which multidimensional arrays are represented makes them quite flexible. In fact, the rows in an array are not required to be the same length. For example,

Click here to view code image

```
int[][] b = {{1, 2}, {3, 4, 5}};
```

creates integer array `b` with two elements (determined by the number of nested array initializers) that represent the rows of the two-dimensional array. Each element of `b` is a reference to a one-dimensional array of `int` variables. The `int` array for row `0` is a one-dimensional array with two elements (`1` and `2`), and the `int` array for row `1` is a one-dimensional array with three elements (`3`, `4` and `5`).

Creating Two-Dimensional Arrays with Array-Creation Expressions

A multidimensional array with the same number of columns in every row can be created with an array-creation expression. For example, the following line declares array `b` and assigns it a reference to a three-by-four `int` array:

```
int[][] b = new int[3][4];
```

The elements of a multidimensional array are initialized when the array object is created.

A multidimensional array in which each row has a different number of columns can be created as follows:

Click here to view code image

```
int[][] b = new int[2][];   // create 2 rows
b[0] = new int[5]; // create 5 columns for row 0
b[1] = new int[3]; // create 3 columns for row 1
```

The preceding statements create a two-dimensional array with two rows. Row `0` has five columns, and row `1` has three columns.

Two-Dimensional Array Example: Displaying Element Values

Figure 6.10 demonstrates initializing two-dimensional arrays with array initializers and using nested `for` loops to **traverse the arrays** (i.e., manipulate every element of each array). Class `InitArray`'s `main` declares two arrays. The declaration of `array1` (line 7) uses nested array initializers of the same length to initialize the first row to the values 1, 2 and 3, and the second row to the values 4, 5 and

6. The declaration of `array2` (line 8) uses nested initializers of different lengths—the first row is initialized to two elements with the values 1 and 2; the second row is initialized to one element with the value 3; and the third row is initialized to three elements with the values 4, 5 and 6.

Click here to view code image

```
1  // Fig. 6.10: InitArray.java
2  // Initializing two-dimensional arrays.
3
4  public class InitArray {
5     // create and output two-dimensional arrays
6     public static void main(String[] args) {
7        int[][] array1 = {{1, 2, 3}, {4, 5, 6}};
8        int[][] array2 = {{1, 2}, {3}, {4, 5, 6}};
9
10       System.out.println("Values in array1 by row are");
11       outputArray(array1); // displays array1 by row
12
13       System.out.printf("%nValues in array2 by row are%n");
14       outputArray(array2); // displays array2 by row
15    }
16
17    // output rows and columns of a two-dimensional array
18    public static void outputArray(int[][] array) {
19       // loop through array's rows
20       for (int row = 0; row < array.length; ++row) {
21          // loop through columns of current row
22          for (int column = 0; column < array[row].length; ++column) {
23             System.out.printf("%d  ", array[row][column]);
24          }
25
26          System.out.println();
27       }
28    }
29 }
```

```
Values in array1 by row are
1  2  3
4  5  6

Values in array2 by row are
1  2
3
4  5  6
```

Fig. 6.10 | Initializing two-dimensional arrays.

Lines 11 and 14 call method `outputArray` (lines 18–28) to output the elements of `array1` and `array2`, respectively. Method `outputArray`'s parameter—`int[][] array`—indicates that the method receives a two-dimensional array. The nested `for` statement (lines 20–27) outputs the rows of a two-dimensional array. In the loop-continuation condition of the outer `for` statement (line 20), the expression `array.length` determines the number of rows in the array. In the inner `for` statement, the line 22 expression `array[row].length` determines the number of columns in the current row of the array. The inner `for` statement's condition enables the loop to determine the number of columns in each row.

Nested Enhanced `for` Loops

We used counter-controlled loops in this example to demonstrate array access expressions with two sets of square brackets (e.g., `array[row][column]` in line 23). You can also implement lines 20–27 using enhanced `for` statements:

Click here to view code image

```
// loop through array's rows using enhanced for loop
for (int[] row : array) {
   // loop through columns of current row using enhanced for loop
   for (int element : row) {
      System.out.printf("%d  ", element);
   }

   System.out.println();
}
```

In the outer loop, the variable `row` is a one-dimensional `int` array representing the current row of the two-dimensional `array` object—you can read this `for` header as, "for each `row` in `array`." In the inner loop, the variable `element` is an `int` representing an element from the current `row`—you can read this `for` header as, "for each `element` in the current `row`." When the inner loop finishes processing the current `row`, the outer loop outputs a newline then processes the next `row` (if any).

Common Multidimensional-Array Manipulations

Let's consider several other common manipulations using the three-row and four-column array named `a` from the diagram at the beginning of Section 6.16. The following `for` statement sets all the elements in row `2` of array `a` to zero:

Click here to view code image

```
for (int column = 0; column < a[2].length; ++column) {
   a[2][column] = 0;
}
```

We specified row `2`; therefore, we know that the first index is always `2` (`0` is the first row, and `1` is the second row). This `for` loop varies only the second index (i.e., the column index). If row `2` of array `a` contains four elements, then the preceding `for` statement is equivalent to the assignment statements

```
a[2][0] = 0;
a[2][1] = 0;
a[2][2] = 0;
a[2][3] = 0;
```

The following nested enhanced `for` statement totals the values of all the elements in array `a`:

```
int total = 0;
for (int[] row : a) {
   for (int element : row) {
      total += element;
   }
}
```

These nested `for` statements total the array elements one row at a time:

- The outer `for` statement initially selects row 0, so the first row's elements can be totaled by the inner `for` statement.
- Next, the outer `for` selects row `1` so the second row can be totaled.
- Then, the outer `for` selects row `2` so the third row can be totaled.

The variable `total` can be displayed when the outer `for` statement terminates.

Generative AI

1 Prompt genAIs to write nested `for` loops that determine and display the largest value in the two-dimensional array of `double`s named `values`. Do the genAIs use nested counting `for` loops or

nested enhanced `for` loops?

6.17 Variable-Length Argument Lists

With **variable-length argument lists**, you can create methods that receive an unspecified number of arguments. In a method's parameter list, a type followed by an **ellipsis (...)** indicates the method can receive a variable number of arguments of that type. This use of the ellipsis may occur only once in a parameter list. Such a parameter must be placed at the end of the parameter list. While you can use method overloading and array passing to accomplish much of what variable-length argument lists do, using an ellipsis in a method's parameter list is more concise.

Figure 6.11 demonstrates the method `average` (lines 6–15), which receives a variable-length sequence of `double`s. Java treats the variable-length argument list as an array. So, the method body manipulates the parameter `numbers` as an array of `double`s. Lines 10–12 use the enhanced `for` loop to calculate the total of the `double`s in the array. Line 14 uses `numbers.length` to obtain the size of the `numbers` array for use in the averaging calculation. Lines 27, 29 and 31 in `main` call method `average` with two, three and four arguments, respectively. Method `average` has a variable-length argument list (line 6), so it can average as many `double` arguments as the caller passes. The output shows that each call to the method `average` returns the correct value.

Click here to view code image

```
1  // Fig. 6.11: VarargsTest.java
2  // Using variable-length argument lists.
3
4  public class VarargsTest {
5     // calculate average
6     public static double average(double... numbers) {
7        double total = 0.0;
8
9        // calculate total using the enhanced for statement
10       for (double d : numbers) {
11          total += d;
12       }
13
14       return total / numbers.length;
15    }
16
17    public static void main(String[] args) {
```

```
        double d1 = 10.0;
        double d2 = 20.0;
        double d3 = 30.0;
        double d4 = 40.0;

        System.out.printf("d1 = %.1f%nd2 = %.1f%nd3 = %.1f%nd4 = %.1f%n%n",
           d1, d2, d3, d4);

        System.out.printf("Average of d1 and d2 is %.1f%n",
           average(d1, d2));
        System.out.printf("Average of d1, d2 and d3 is %.1f%n",
           average(d1, d2, d3));
        System.out.printf("Average of d1, d2, d3 and d4 is %.1f%n",
           average(d1, d2, d3, d4));
     }
  }
```

```
d1 = 10.0
d2 = 20.0
d3 = 30.0
d4 = 40.0

Average of d1 and d2 is 15.0
Average of d1, d2 and d3 is 20.0
Average of d1, d2, d3 and d4 is 25.0
```

Fig. 6.11 | Using variable-length argument lists.

Generative AI

1 If you call Fig. 6.11's `average` method with no arguments it would attempt to calculate `0.0/0`, which does not have a numeric representation. Prompt genAIs with Fig. 6.11's `average` method and ask them to modify the Java code so `average` requires at least one argument—calling it with no arguments should result in a compilation error. Call the generated `average` method with no arguments to confirm that the compiler generates an error.

6.18 Command-Line Arguments

You can pass arguments from the command line to a program via method `main`'s `String[]` parameter, which is named `args` by convention. Typical uses of command-line arguments include passing configuration options, data, filenames and folder paths to programs. When you execute a program with the `java` command,

the JVM passes the command-line arguments (that appear after the class name) to the program's `main` method as `String`s in the array `args`. The array's `length` attribute stores the number of command-line arguments.

In Fig. 6.12, we use command-line arguments to determine an array's size, its first element value and the increment for calculating its remaining element values. The command

```
java InitArray 5 0 4
```

passes three arguments—`5`, `0` and `4`—to the `InitArray` program. Command-line arguments are separated by whitespace. When this command executes, `InitArray`'s `main` method's `args` parameter receives a three-element array (`args.length` is `3`) in which:

- `args[0]` contains the `String "5"`,
- `args[1]` contains the `String "0"` and
- `args[2]` contains the `String "4"`.

The program determines how to use these arguments. For this example, we convert them to `int` values and use them to initialize an array. When the program executes, if `args.length` is not `3`, the program prints an error message and terminates by returning from `main` immediately (lines 7–12). Otherwise, lines 15–32 initialize and display the array based on the command-line arguments.

Click here to view code image

```
1  // Fig. 6.12: InitArray.java
2  // Initializing an array using command-line arguments.
3
4  public class InitArray {
5     public static void main(String[] args) {
6        // check number of command-line arguments
7        if (args.length != 3) {
8           System.out.println("""
9              Error: Please re-execute the program and include
10             an array size, initial value and increment.""");
11          return;
12       }
13
14       // get array size from first command-line argument
15       int arrayLength = Integer.parseInt(args[0]);
16       int[] array = new int[arrayLength];
17
18       // get initial value and increment from command-line arguments
```

```
19          int initialValue = Integer.parseInt(args[1]);
20          int increment = Integer.parseInt(args[2]);
21
22          // calculate value for each array element
23          for (int counter = 0; counter < array.length; ++counter) {
24             array[counter] = initialValue + increment * counter;
25          }
26
27          System.out.printf("%s%8s%n", "Index", "Value");
28
29          // display array index and value
30          for (int counter = 0; counter < array.length; ++counter) {
31             System.out.printf("%5d%8d%n", counter, array[counter]);
32          }
33       }
34    }
```

```
> java InitArray
Error: Please re-execute the program and include
an array size, initial value and increment.""");
```

```
> java InitArray 5 0 4
Index   Value
    0       0
    1       4
    2       8
    3      12
    4      16
```

```
> java InitArray 8 1 2
Index   Value
    0       1
    1       3
    2       5
    3       7
    4       9
    5      11
    6      13
    7      15
```

Fig. 6.12 | Initializing an array using command-line arguments.

Line 15 gets `args[0]`—a `String` that specifies the array size—and converts it to an `int` value that the program uses to create the array in line 16. The **`Integer` class `static` method `parseInt`** converts its `String` argument to an `int`. This method can fail—in

Generative AI question 1, you'll prompt genAIs to deal with this possibility.

Lines 19–20 convert the `args[1]` and `args[2]` command-line arguments to `int` values and store them in `initialValue` and `increment`, respectively. Lines 23–25 calculate each array element's value. Lines 27–32 display the array's contents.

In the three sample executions:

- The first shows that the program received an insufficient number of command-line arguments.
- The second uses the command-line arguments 5, 0 and 4 to specify the array's size (5), the first element's value (0) and the increment for calculating each subsequent element's value (4). The output shows that these values create an array containing 0, 4, 8, 12 and 16.
- The third uses the command-line arguments 8, 1 and 2 to produce an array whose 8 elements are the nonnegative odd integers from 1 to 15.

Generative AI

1 Prompt genAIs with Fig. 6.12's code and ask them to handle the exception that would occur if the user enters a non-integer command-line argument.

6.19 Class `Arrays`

The **`Arrays` class** helps you avoid reinventing the wheel by providing `static` methods for common array manipulations, such as:

- **`sort`** for sorting an array—that is, arranging elements into ascending or descending order,
- **`binarySearch`** for searching a sorted array—that is, determining whether an array contains a specific value and, if so, where the value is located,
- **`equals`** for determining whether two arrays have the same

contents,

- **`fill`** for filling an array's elements with the same value, and
- **`toString`** for creating a `String` representation of an array's contents without having to iterate through the elements using a loop.

These and many other `Arrays` methods are overloaded for arrays of primitive-type and reference-type elements. Figure 6.13 uses `Arrays` methods `sort`, `binarySearch`, `equals`, `fill` and `toString`, and shows how to copy arrays with the **`System` class's `static` `arraycopy` method**.

Click here to view code image

```
1  // Fig. 6.13: ArrayManipulations.java
2  // Arrays class methods and System.arraycopy.
3  import java.util.Arrays;
4
5  public class ArrayManipulations {
6     public static void main(String[] args) {
7        // sort doubles into ascending order
8        double[] doubles = {8.4, 9.3, 0.2, 7.9, 3.4};
9        Arrays.sort(doubles);
10       System.out.printf("doubles: %s%n", Arrays.toString(doubles));
11
12       // fill 10-element array with 7s
13       int[] filledInts = new int[10];
14       Arrays.fill(filledInts, 7);
15       System.out.printf("filledInts: %s%n", Arrays.toString(filledInts));
16
17       // copy array ints into array intsCopy
18       int[] ints = {1, 2, 3, 4, 5, 6};
19       int[] intsCopy = new int[ints.length];
20       System.arraycopy(ints, 0, intsCopy, 0, ints.length);
21       System.out.printf("ints: %s%n", Arrays.toString(ints));
22       System.out.printf("intsCopy: %s%n", Arrays.toString(intsCopy));
23
24       // compare ints and intsCopy for equality
25       boolean b = Arrays.equals(ints, intsCopy);
26       System.out.printf("%nints %s intsCopy%n",
27           (b ? "equals" : "does not equal"));
28
29       // compare ints and filledInts for equality
30       b = Arrays.equals(ints, filledInts);
31       System.out.printf("ints %s filledInts%n",
32           (b ? "equals" : "does not equal"));
33
34       // search ints for the value 5
```

```
35          int location = Arrays.binarySearch(ints, 5);
36
37          if (location >= 0) {
38             System.out.printf("Found 5 at element %d in ints%n", location);
39          }
40          else {
41             System.out.println("5 not found in ints");
42          }
43
44          // search intArray for the value 8763
45          location = Arrays.binarySearch(ints, 8763);
46
47          if (location >= 0) {
48             System.out.printf(
49                "Found 8763 at element %d in ints%n", location);
50          }
51          else {
52             System.out.println("8763 not found in ints");
53          }
54       }
55    }
```

```
doubles: [0.2, 3.4, 7.9, 8.4, 9.3]
filledInts: [7, 7, 7, 7, 7, 7, 7, 7, 7, 7]
ints: [1, 2, 3, 4, 5, 6]
intsCopy: [1, 2, 3, 4, 5, 6]

ints equals intsCopy
ints does not equal filledInts
Found 5 at element 4 in ints
8763 not found in ints
```

Fig. 6.13 | `Arrays` class methods and `System.arraycopy`.

Sorting an Array's Elements

In `main`, line 9 sorts the elements of array `doubles`. The `static` method `sort` of class `Arrays` orders the array's elements in ascending order by default. Overloaded versions of `sort` allow you to sort a specific range of elements within the array. Line 10 displays the sorted array. **`Arrays` `static` method `toString`** returns a string representation of an array's elements in a comma-separated list delimited by square brackets.

Filling an Array's Elements with a Specified Value

Line 14 calls `static` method `fill` of class `Arrays` to populate the

10 elements of array `filledInts` with `7`s. Overloaded versions of `fill` allow you to populate a specific range of elements with the same value. Line 15 displays the filled array.

Copying Arrays

Line 20 copies the elements of `ints` into `intsCopy`. The first argument (`ints`) passed to `System` method `arraycopy` is the array from which elements are to be copied. The second argument (`0`) is the index specifying the starting point in the range of elements to copy from the array. This value can be any valid array index. The third argument (`intsCopy`) specifies the destination array that will store the copy. The fourth argument (`0`) specifies the index in the destination array where the first copied element should be stored. The last argument specifies the number of elements to copy from the array in the first argument. In this case, we copy all the elements in the array.

Comparing Arrays

Lines 25 and 30 call `static` method `equals` of class `Arrays` to determine whether all the elements of two arrays are equivalent. If the arrays contain the same elements in the same order, the method returns `true` (line 25); otherwise, it returns `false` (line 30). When comparing array contents, always use `Arrays.equals(array1, array2)`, which compares the two arrays' contents, rather than `array1.equals(array2)`, which compares whether `array1` and `array2` refer to the same array object.

Searching Arrays for a Specified Value

Lines 35 and 45 call `Arrays` `static` method `binarySearch` to perform a binary search on `ints`, using the second argument (`5` and `8763`, respectively). The value you are searching for is known as the **key** or **search key**. If the key is found (line 35), `binarySearch` returns the element's index; otherwise, `binarySearch` returns a negative value (line 45). Passing an unsorted array to `binarySearch` is a logic error—the value returned is undefined.

Class `Arrays` Method `parallelSort`

The `Arrays` class has several "parallel" methods that can use multi-core hardware to significantly improve performance. `Arrays` method `parallelSort` can sort large arrays more efficiently on

multi-core systems. In Section 18.2, we create a 100,000,000-element array and use features of the Date/Time API to compare how long it takes to sort the array with `sort` and `parallelSort`.

Generative AI

1 Prompt genAIs to write a tutorial showing how to use the top methods of Java class `Arrays`.

6.20 Objects-Natural Case Study: Intro to Collections and Class `ArrayList`

We now continue our objects-natural approach with a case study on creating and manipulating objects of the **Java Collection Framework's `ArrayList<E>` collection class**. The Java API provides many predefined collections (that is, data structures) that store groups of related objects in memory. These classes provide efficient, proven methods that organize, store and retrieve your data portably without requiring knowledge of how the data is being stored. This reduces program development time and helps create more robust applications. Chapter 12, Generic Collections, covers the Java Collection Framework in detail.

You've used arrays to store sequences of objects. Arrays are fixed in size, so they cannot automatically change their size at execution time to accommodate additional or fewer elements. The collection class `ArrayList<E>` (package `java.util`) provides a convenient solution to this problem. The `E` is a placeholder for the `ArrayList`'s element type. When declaring a new `ArrayList`, you replace `E` with the element type. For example,

```
ArrayList<String> list;
```

declares `list` as an `ArrayList` collection that can store only `String`s. Once the collection's element type is specified, the compiler ensures all operations using that collection are **type-safe**. For example, the compiler ensures that all elements you add to an `Array-List<String>` are `String`s; otherwise, compilation errors occur. As with arrays, collection indices start at zero. In fact, `ArrayList` is implemented using a conventional array behind the

scenes. The following table shows some key methods of class `ArrayList<E>`:

Method	Description
`Add`	Adds an element to the end of an `ArrayList` or at a specific index in an `ArrayList`.
`AddFirst`	Adds an element to the beginning of an `ArrayList`.
`AddLast`	Adds an element to the end of an `ArrayList`.
`Clear`	Removes all the elements from an `ArrayList`.
`Contains`	Returns a `boolean` indicating whether an `ArrayList` contains the specified value.
`Get`	Returns the element at the specified index.
`GetFirst`	Returns the first element of an `ArrayList`.
`GetLast`	Returns the last element of an `ArrayList`.
`IndexOf`	Returns the index of the first occurrence of the specified value in an `ArrayList`.
`Remove`	Removes the specified value's first occurrence or the element at the specified index.
`RemoveFirst`	Removes the first element of an `ArrayList`
`RemoveLast`	Removes the last element of an `ArrayList`
`Size`	Returns the number of elements stored in an `ArrayList`.

Boxing and Unboxing

Classes with type placeholders that can be used with any type are called **generic classes**. Only reference types can be used to declare variables and create objects of generic classes. Java provides a mechanism—known as **boxing**—that allows primitive values to be wrapped as objects for use with generic classes. For example, an `ArrayList<Integer>` can store only `Integer` objects. When you place an `int` value into an `ArrayList<Integer>`, the `int` value is boxed (wrapped) as an `Integer` object, and when you get an `Integer` object from an `ArrayList<Integer>` then assign the object to an `int` variable, the `int` value inside the object is **unboxed** (unwrapped). This boxing/unboxing process is performed transparently to the programmer.

Demonstrating an `ArrayList<String>`

Figure 6.14 demonstrates some common `ArrayList` capabilities. Line 8 creates a new empty `ArrayList` of `String`s with an initial capacity of three elements. The capacity indicates how many items the `ArrayList` can hold without growing. When the `ArrayList`

grows, it must create a larger internal array and copy each element to the new array. This operation is time-consuming. It would be inefficient for the `ArrayList` to grow by one element each time an element is added. Instead, it grows by approximately 50% only when an element is added and the number of elements is equal to the capacity—i.e., there's no space for the new element. If you do not pass a capacity when you create the `ArrayList`, the default initial capacity is 10. We chose the initial capacity 3 (line 8) in this example to show that the `ArrayList` grows once we attempt to add an element beyond its initial capacity.

Click here to view code image

```
1  // Fig. 6.14: ArrayListCollection.java
2  // Generic ArrayList<E> collection demonstration.
3  import java.util.ArrayList;
4
5  public class ArrayListCollection {
6     public static void main(String[] args) {
7        // create a new ArrayList of Strings with an initial capacity of 3
8        var items = new ArrayList<String>(3);
9        System.out.printf(
10          "After creating list with initial capacity of 3, size is: %d%n",
11          items.size());
12
13       items.add("red"); // append an item to the list
14       items.add(0, "yellow"); // insert "yellow" at index 0
15
16       System.out.print(
17          "Display list contents with counter-controlled for loop:");
18
19       // display the colors in the list
20       for (int i = 0; i < items.size(); ++i) {
21          System.out.printf(" %s", items.get(i));
22       }
23
24       // display colors using enhanced for in the display method
25       display(items,
26          "%nDisplay list contents with enhanced for loop:");
27       System.out.printf("list size: %d%n", items.size());
28
29       items.add("green"); // add "green" to the end of the list
30       items.add("yellow"); // add "yellow" to the end of the list
31       display(items, "\nList with two new elements:");
32       System.out.println("List has grown beyond initial capacity of 3");
33       System.out.printf("list size: %d%n", items.size());
34
35       items.remove("yellow"); // remove the first "yellow"
36       display(items, "\nRemove first instance of yellow:");
```

```
37          System.out.printf("list size: %d%n", items.size());
38
39          items.remove(1); // remove item at index 1
40          display(items, "\nRemove second list element (green):");
41          System.out.printf("list size: %d%n", items.size());
42
43          // check if a value is in the List
44          System.out.printf("\n\"red\" is %sin the list%n",
45             items.contains("red") ? "": "not ");
46
47          // get the index of an item in the list
48          System.out.printf("\"red\" is located at index: %d%n",
49             items.indexOf("red"));
50
51          // clear the ArrayList and display its size
52          items.clear();
53          System.out.printf("\nSize after clear: %s%n", items.size());
54       }
55
56       // display the ArrayList's elements on the console
57       public static void display(ArrayList<String> items, String header) {
58          System.out.printf(header); // display header
59
60          // display each element in items with enhanced for
61          for (String item : items) {
62             System.out.printf(" %s", item);
63          }
64
65          System.out.println();
66       }
67    }
```

```
After creating list with initial capacity of 3, size is: 0
Display list contents with counter-controlled for loop: yellow red
Display list contents with enhanced for loop: yellow red
list size: 2

List with two new elements: yellow red green yellow
List has grown beyond initial capacity of 3
list size: 4

Remove first instance of yellow: red green yellow
list size: 3

Remove second list element (green): red yellow
list size: 2

"red" is in the list
"red" is located at index: 0
```

```
Size after clear: 0
```

Fig. 6.14 | Generic `ArrayList<E>` collection demonstration.

The **`add` method** adds elements to the `ArrayList` (lines 13–14). The version with one argument (line 13) appends it to the end of the `ArrayList`. The version with two arguments (line 14) inserts a new element at the specified position. The first argument is an index. The second argument is the value to insert at that index. The indices of all subsequent elements are incremented by one. Inserting an element is usually slower than adding an element to the end of the `ArrayList`.

Lines 20–22 display the `ArrayList`'s items. The **`size` method** returns the `Array-List`'s current number of elements—unlike arrays, `ArrayList`s do not have a `length` instance variable. The **`get` method** (line 21) obtains the element at the specified index. Lines 25–26 display the elements again by invoking method `display` (lines 57–66), which uses an enhanced `for` loop to iterate through the `ArrayList`'s elements, just as we did with built-in arrays earlier in this chapter. Then, line 27 displays the `ArrayList`'s current size. Lines 29–30 add two more elements to the `ArrayList`, then line 31 displays the elements again to confirm that the two elements were added to the end of the collection, and line 33 displays the `ArrayList`'s new size. Note that the size is now larger than the `ArrayList`'s initial capacity.

The **`remove` method** call in line 35 deletes the first element with a specific value (`"yellow"`). If no such element is in the `ArrayList`, `remove` does nothing. Lines 36–37 display the `ArrayList`'s current elements and size to show that the first `"yellow"` was removed and the size decreased by one. Line 39 uses an overloaded version of `remove` that deletes the element at the specified index (in this case, `1`). When an element is deleted, the indices of any elements after the removed element decrease by one—a time-consuming operation. Lines 40–41 display the `ArrayList`'s current elements and size to show that `"green"` was removed and the size decreased by one again.

Lines 44–45 use the **`contains` method** to check whether `"red"` is in the `ArrayList`. The `contains` method returns `true` if the element is found in the `ArrayList`; otherwise, it returns `false`. Lines 48–49 use the **`indexOf` method** to get the index of `"red"` in the `ArrayList` or `-1` if `"red"` is not found. Methods `contains` and `indexOf` compare their arguments to each `ArrayList` element in order—known as a linear search—so using them on large

`ArrayList`s can be inefficient.

Line 52 removes all the `ArrayList`'s remaining elements by calling the **`clear` method**. Then, line 53 shows that the `ArrayList`'s size is now 0.

Diamond (<>) Notation for Creating an Object of a Generic Class

Figure 6.14's `ArrayList` is a local variable, so we used local-variable type inference in line 8 to determine the type of variable `items` from its initializer. In older code, line 8 might appear as:

Click here to view code image

```
ArrayList<String> items = new ArrayList<>();
```

The **diamond (<>) notation** in the `new` expression tells the compiler to infer the `ArrayList` object's element type (`String`) from the declaration of variable `items`. The preceding statement shortens statements like

Click here to view code image

```
ArrayList<String> items = new ArrayList<String>();
```

in which `ArrayList<String>` appears in both the variable declaration and the `new` expression.

Generative AI

1 Prompt genAIs with the code in Fig. 6.14 and ask them to rewrite the code to remove the calls to `displayArray` and replace them with calls to `System.out.printf` that implicitly use `ArrayList`'s `toString` method. Run the updated code and compare the output to Fig. 6.14's output.

6.21 Wrap-Up

This chapter began our introduction to data structures, using arrays to store data in and retrieve data from lists and tables of values. We discussed the differences between primitive-type variables, which store values, and reference-type variables, which store references to objects elsewhere in memory. We introduced arrays, discussing how to declare, create and initialize them, and explaining how to access

and manipulate their elements.

You saw that array elements are initialized by default and that you can use array initializers to provide custom initializers for them. You also used loops to assign calculated values to array elements. We showed common array manipulations, including totaling array elements, presenting array data visually with a bar chart, and using array elements as counters to summarize die-roll frequencies and survey results. We used the `try` statement to handle an `ArrayIndexOutOfBoundsException` when a program attempted to access an array element outside an array's bounds.

You iterated through arrays using counting `for` loops. We also used the enhanced `for` loop to iterate over an array's elements without a counter, eliminating various counting-error possibilities and preventing attempts to access elements outside an array's bounds. We passed arrays and individual array elements to methods to demonstrate the difference between pass-by-value and pass-by-reference.

We introduced multidimensional arrays and used two-dimensional arrays to represent tables containing rows and columns of data. You declared, created and manipulated two-dimensional arrays. You iterated through two-dimensional arrays using nested counting `for` loops and nested enhanced `for` loops. We showed how to write methods that use variable-length argument lists and how to read arguments passed to a program from the command line. You performed various array manipulations with the `Arrays` class's `static` methods.

The Objects-Natural Case Study introduced the `ArrayList<E>` generic collection, which provides the functionality and performance of arrays, as well as dynamic resizing to accommodate more elements. You used the `add` and `remove` methods to add new items to and remove items from an `ArrayList`. We showed `ArrayList` method `size` returns the current number of items in the `ArrayList`. We also used `contains` to determine whether an item was in the `ArrayList`.

We continue our coverage of data structures in Chapter 12, Generic Collections, which introduces several collections from the Java Collections Framework and additional methods of class `Arrays`. In Chapter 14, Functional Programming with Lambdas & Streams, we introduce Java's functional programming capabilities, which enable you to write more concise code that can have fewer errors and that can be easier to read, debug and modify. Chapter 14 is keyed to appropriate earlier sections of the book so that you can

conveniently use lambdas and streams sooner if you'd like. Now that you've completed Chapter 6, you can read Sections 14.1–14.7, which introduce the lambdas and streams concepts and use them to rework examples from Chapters 5 and 6. After completing Chapter 14, you'll be able to reimplement many of Chapter 6's examples more concisely and elegantly, and in a way that makes them easier to parallelize to improve performance on today's multi-core systems. In Chapter 18, Concurrency: Platform Threads to Virtual Threads, we'll demonstrate that improved performance by timing various operations using features of the Date/Time API.

In the next chapter, we discuss class `String` and its methods. We introduce regular expressions for pattern matching in strings and use them to validate user input.

Special Section: Generative AI Exercises

For each of the following questions, try multiple genAIs to determine which performs best. Always verify the results. Keep in mind that they sometimes make mistakes and even hallucinate, creating unreliable information.

6.1 ***(Research the Next Version of Java)*** New Java versions are released every March and September. Prompt generative AIs with the current version of Java and ask them to write a tutorial covering the significant new language and library features scheduled to be released in the next version.

6.2 ***(One-Dimensional Array Animation Demos)*** Prompt generative AIs to locate animations that demonstrate iterating through a one-dimensional array.

6.3 ***(Two-Dimensional Array Animation Demos)*** Prompt generative AIs to locate animations that demonstrate iterating through a two-dimensional array.

6.4 ***(One-Dimensional Array Practice Exercises)*** Prompt generative AIs to create practice exercises with answers for one-dimensional Java array manipulations.

6.5 ***(Two-Dimensional Array Practice Exercises)*** Prompt generative AIs to create practice exercises with answers for two-dimensional Java array manipulations.

6.6 ***(What Does This One-Dimensional Array Code Do?)*** Prompt generative AIs to create cryptic code examples for one-dimensional

Java array manipulations so you can study them and determine what they do.

6.7 ***(What Does This Two-Dimensional Array Code Do?)*** Prompt generative AIs to create cryptic code examples for two-dimensional Java array manipulations so you can study them and determine what they do.

6.8 ***(Failing Fast vs. Exception Handling)*** Some developers believe that during program development, it's better to allow a program to fail when an exception occurs (known as "failing fast") rather than catching and handling the exception. Prompt generative AIs to discuss the benefits and disadvantages of failing fast vs. exception handling.

6.9 ***(Improving Fault Tolerance)*** Prompt generative AIs with a program and ask them to improve the code's fault tolerance by adding exception handling for methods that throw exceptions.

6.10 ***(Making a Difference)*** Prompt generative AIs for suggested Java programs and projects you can build to "make a difference" in the world.

6.11 ***(Checking Java Idiom)*** Prompt generative AIs with code you write and ask them to evaluate the code for proper idiom in the context of the latest Java version and what you've learned so far in this book.

6.12 ***(Primitive vs. Reference Types)*** Prompt generative AIs to write a tutorial on primitive types vs. reference types in Java.

6.13 ***(Passing Arguments to Methods)*** Prompt generative AIs to write a tutorial comparing pass-by-value and pass-by-reference in Java.

6.14 ***(Modernizing Java Code)*** Because of the vast amounts of Java legacy code that have been written since Java appeared in 1995, many developers are concerned with modernizing their organizations' code bases. Prompt generative AIs to write a tutorial on modernizing Java code.

Strings, NLP and Regex:

Generative AI Foundations

Objectives

In this chapter, you'll:

- Create and manipulate immutable character-string objects of class `String`.
- Create and manipulate mutable character-string objects of class `StringBuilder`.
- Test characters for various attributes using `static` methods of class `Character`.
- Tokenize `String`s.
- Use generative AI to leverage your Java learning experience.
- Be introduced to natural language processing—a key foundation of generative AI.
- See an objects-natural case study on using regular expressions to recognize patterns in text for data validation, transformation and extraction—operations commonly used to prepare text to train language models used with generative AI.
- See an objects-natural case study on pMa5tfEKwk59d TvC04Ft1IFQz9mEXnkfYXZwxk4ujGE= (see Section 7.9).

Outline

7.1 Introduction

7.2 Fundamentals of Characters and Strings

7.3 Class `String`

- 7.3.1 Creating `String` Objects
- 7.3.2 `String` Methods `length`, `charAt` and `getChars`
- 7.3.3 Comparing `String`s
- 7.3.4 Locating Characters and Substrings in `String`s
- 7.3.5 Extracting Substrings from `String`s
- 7.3.6 Concatenating `String`s

7.3.7 Miscellaneous `String` Methods
7.3.8 `String` Method `valueOf`
7.4 Class `StringBuilder`
7.4.1 Creating `StringBuilder` Objects
7.4.2 Methods `length`, `capacity`, `setLength` and `ensureCapacity`
7.4.3 Methods `charAt`, `setCharAt`, `getChars` and `reverse`
7.4.4 `append` Methods
7.4.5 Insertion and Deletion Methods
7.5 Class `Character`
7.5.1 `Character` Testing and Case Conversion Methods
7.5.2 Character/Digit Conversions
7.5.3 Other `Character` Methods
7.6 Tokenizing `String`s
7.7 Intro to Natural Language Processing (NLP)—at the Root of Generative AI
7.8 Objects-Natural Case Study: Intro to Regular Expressions in NLP
7.8.1 Matching Complete `String`s to Patterns
7.8.2 Replacing Substrings
7.8.3 Searching for Matches with Classes `Pattern` and `Matcher`
7.8.4 Simple Data Wrangling Steps Used to Prepare Text for Training NLP and Generative AI Models
7.9 Objects-Natural Security Case Study: pMa5tfEKwk59dTvC04Ft1IFQz9mEX nkfYXZwxk4ujGE=
7.10 Wrap-Up

7.1 Introduction

This chapter discusses Java's string- and character-processing capabilities. The techniques presented here are appropriate for validating program input, displaying information to users and performing many common text-based manipulations. You saw

several string-processing capabilities in earlier chapters. This chapter discusses in detail the `java.lang` package's `String`, `StringBuilder` and `Character` classes. You'll continue using generative AI to enhance your learning experience.

Introduction to Natural Language Processing (NLP)

Section 7.7 introduces natural language processing (NLP)—an important data science and AI topic. NLP is a key foundational technology for generative AI. NLP helps computers understand, analyze and process text. While writing this book, we used the paid NLP tool Grammarly[1] to help tune the writing and ensure the text's readability for a broad audience. We also used various generative AIs' NLP capabilities to evaluate the chapters for breadth and depth of coverage, verify the content and suggest potential improvements.

1. Grammarly has free and paid versions (`https://www.grammarly.com`). They provide free plug-ins you can use in several popular web browsers.

Objects-Natural Case Study: Using Regular Expressions to Search Strings for Patterns, Validate Data and Replace Substrings

We continue our objects-natural approach with two case studies. Section 7.8 introduces regular expressions (regexes for short), which are crucial in today's data-rich applications for

- cleaning and preparing data for analysis and large language model (LLM) training—a process called **data wrangling** or **data munging**,
- data mining—such as locating names, organizations, URLs, email addresses and phone numbers in large bodies of text—and
- transforming data to other formats—such as converting tab-delimited data to comma-separated-values (CSV) data.

Via class `String` and classes `Matcher` and `Pattern` from the `java.util.regex` package, we'll use regular expressions to match patterns in text. Earlier chapters mentioned the importance of validating user input, especially in industrial-strength code. The capabilities presented in this chapter are frequently used to validate data and in

- natural-language processing,
- preparing data for training generative AIs and
- enabling generative AIs to create responses to your prompts.

Objects-Natural Security Case Study: pMa5tfEKwk59dTvC04Ft1IFQz9mEX nkfYXZwxk4ujGE=

The title of our second objects-natural case study is not a mistake. See Section 7.9 for what these seemingly random characters, digits and symbols mean.

7.2 Fundamentals of Characters and Strings

Characters are the fundamental building blocks of Java code. A **character literal** is an integer value represented as a character in single quotes. For example, `'z'` represents the letter `z`'s integer value, and `'\t'` represents a tab character's integer value. A character literal's value is that character's integer value in the **Unicode character set**[2] (introduced in Chapter 1). Every program is composed of a sequence of characters that—when grouped meaningfully—are interpreted by the Java compiler as instructions to accomplish a task.

2. "List of Unicode characters." Wikipedia, last modified June 20, 2024. Accessed September 6, 2024. https://en.wikipedia.org/wiki/List_of_Unicode_characters.

Recall from Section 2.2 that a string is a sequence of characters treated as a single unit that may include letters, digits and various **special characters**, such as `+`, `-`, `*`, `/` and `$`. A string is an object of class `String`. **String literals** (stored in memory as `String` objects) are written as a sequence of characters in double quotation marks, as in:

Click here to view code image

```
"John Q. Huang"            (a name)
"99999 Main Street"        (a street address)
"Waltham, Massachusetts"   (a city and state)
```

```
"(201) 555-1212"          (a telephone number)
```

You've assigned string literals to `String` variables. The following declaration initializes the `String` variable `color` to refer to a `String` object containing `"blue"`:

```
String color = "blue";
```

String Objects Cannot Be Modified

Java treats all string literals with the same contents as one `String` object with many references to it to conserve memory. None of those references can be used to modify the `String` because `String` objects are **immutable**—the `String` class does not provide methods that modify a `String` object's contents. Each `String` method we demonstrate in this chapter that appears to modify a `String` actually returns a new `String` object containing the modifications, leaving the original unchanged.

Generative AI

1 Prompt genAIs to explain why `String` objects are immutable, but this is not universally true for all data types.

7.3 Class `String`

The next several subsections cover many of class `String`'s capabilities.

7.3.1 Creating `String` Objects

Figure 7.1 shows several ways to create and initialize `String` objects.

Click here to view code image

```
1  // Fig. 7.1: StringConstructors.java
2  // String class constructors.
3
4  public class StringConstructors {
5     public static void main(String[] args) {
6        char[] charArray = {'b', 'i', 'r', 't', 'h', 'd', 'a', 'y'};
7        var s = new String("hello");
```

```
8
9           // use String constructors
10          var s1 = new String(); // ""
11          var s2 = new String(s); // "hello"
12          var s3 = new String(charArray); // "birthday"
13          var s4 = new String(charArray, 5, 3); // "day"
14
15          System.out.printf(
16             "s1 = %s%ns2 = %s%ns3 = %s%ns4 = %s%n", s1, s2, s3, s4);
17       }
18    }
```

```
s1 =
s2 = hello
s3 = birthday
s4 = day
```

Fig. 7.1 | `String` class constructors.

Line 7 uses the `new` operator to create a `String` object. The variable `s` is initialized with the result of the expression

```
new String("hello")
```

which is formally called a **class instance creation expression**.[3] The `new` operator creates an object (that is, an instance) of the specified class—in this case, `String`. The parentheses that follow the class name represent a call to a **constructor**, which is similar to a method but is called implicitly by the `new` operator to initialize an object when it's created. Line 7 calls the **`String` constructor that copies another `String`'s contents** into the new `String` object. We'll discuss constructors in detail in Chapter 8.

3. The Java Language Specification, Java SE 23 Edition. Section 15.9, "Class Instance Creation Expressions." Accessed September 6, 2024. https://docs.oracle.com/javase/specs/jls/se23/html/jls-15.html#jls-15.9.

Line 10 creates a new `String` using the **`String` no-argument constructor**, which creates a `String` object containing no characters (i.e., the empty string, which can also be represented as `""`). Line 11 creates a new `String` object containing a copy of its argument `s`, which contains `"hello"`. Line 12 creates a new `String` object containing a copy of the characters in the **`String` constructor's `char` array argument**.

Line 13 creates a new `String` object using the **`String` constructor that takes a `char` array and two integers as**

arguments. The second argument specifies the starting position (the offset; `String` indices start at 0) from which characters in the array are copied. The third argument specifies the number of characters (the count) to copy from the array. If the offset or the count specified as an argument results in accessing an element outside the array's bounds, an `IndexOutOfBoundsException` occurs.

Generative AI

1 Prompt genAIs to add meaningful comments to the program in Fig. 7.1, then compare the results to the original program.

7.3.2 `String` Methods `length`, `charAt` and `getChars`

`String` methods `length`, **`charAt`** and **`getChars`** return the length of a `String`, obtain the character at a specific location in a `String` and retrieve a set of characters from a `String` as a `char` array, respectively. Figure 7.2 demonstrates these methods.

Click here to view code image

```
1  // Fig. 7.2: StringMiscellaneous.java
2  // This program demonstrates the length, charAt and getChars
3  // methods of the String class.
4
5  public class StringMiscellaneous {
6     public static void main(String[] args) {
7        String s1 = "hello there";
8        char[] charArray = new char[5];
9
10       System.out.printf("s1: %s", s1);
11
12       // test length method
13       System.out.printf("%nLength of s1: %d", s1.length());
14
15       // loop through characters in s1 with charAt and display reversed
16       System.out.printf("%nThe string reversed is: ");
17
18       for (int count = s1.length() - 1; count >= 0; --count) {
19          System.out.printf("%c ", s1.charAt(count));
20       }
21
22       // copy characters from s1 into charArray
23       s1.getChars(0, 5, charArray, 0);
24       System.out.printf("%nThe character array is: ");
```

```
25
26          for (char character : charArray) {
27             System.out.print(character);
28          }
29
30          System.out.println();
31       }
32    }
```

```
s1: hello there
Length of s1: 11
The string reversed is: e r e h t o l l e h
The character array is: hello
```

Fig. 7.2 | `String` methods `length`, `charAt` and `getChars`.

Line 13 uses `String` method `length` to determine the number of characters in `String s1`. Like arrays, `String`s know their own length. However, unlike arrays, you access a `String`'s length using the `String`'s `length` method.

Lines 18–20 print the characters of the `String s1` in reverse order (and separated by spaces). `String` method `charAt` (line 19) returns the character at a specific position in the `String`. Method `charAt` receives an integer argument representing an index and returns the character at that index. Like arrays, the first element of a `String` is at position 0.

Line 23 uses the `String` method `getChars` to copy the characters of a `String` into a character array. The first argument is the starting index from which characters are copied. The second argument is the index that's one past the last character to copy from the `String`. The third argument is the character array into which the characters are copied. The last argument is the starting index where the copied characters are placed in the target character array. Next, lines 26–28 print the `char` array contents one character at a time.

Generative AI

1 Prompt genAIs to write a counting `for` statement equivalent to the enhanced `for` statement in lines 26–28 of Fig. 7.2. Ask them to comment on why the enhanced `for` statement is preferred.

7.3.3 Comparing `String`s

Frequently, information being sorted or searched consists of `String`s that must be compared to place them into order or to determine whether a `String` appears in an array (or other collection). Class `String` provides methods for comparing `String`s, as demonstrated in Figs. 7.3–7.4.

To understand what it means for one `String` to be greater than or less than another, consider the process of alphabetizing a series of last names. No doubt, you'd place "Garcia" before "Tanaka" because the first letter of "Garcia" comes before the first letter of "Tanaka" in the alphabet. The alphabet is more than just a list of 26 letters—it's an ordered list of characters. Each letter occurs in a specific position within the list. Z is more than just a letter of the alphabet—it's specifically the twenty-sixth letter of the alphabet.

How does Java know that one letter "comes before" another? All characters are represented internally as numeric codes in the Unicode character set. When Java compares `String`s, it compares their characters using **lexicographical comparison**—that is, it compares the characters' Unicode numeric values.

Figure 7.3 demonstrates several `String` comparison methods and using the equality operator `==` to compare `String` objects. We broke this example into pieces for discussion purposes. Lines 6–12 create and display several `String`s for use in this example.

Click here to view code image

```
1  // Fig. 7.3: StringCompare.java
2  // Comparing Strings and portions of Strings.
3
4  public class StringCompare {
5     public static void main(String[] args) {
6        String s1 = new String("hello"); // s1 is a copy of "hello"
7        String s2 = "goodbye";
8        String s3 = "Happy Birthday";
9        String s4 = "happy birthday";
10
11       System.out.printf(
12          "s1 = %s%ns2 = %s%ns3 = %s%ns4 = %s%n%n", s1, s2, s3, s4);
13
```

```
s1 = hello
s2 = goodbye
s3 = Happy Birthday
s4 = happy birthday
```

Fig. 7.3 | `String` methods `equals`, `equalsIgnoreCase`, `compareTo` and `regionMatches`.

`String` Method `equals`

Line 15 uses `String` method `equals` to compare `String s1` and the `String` literal `"hello"` for equality:

Click here to view code image

```
14          // test for equality
15          if (s1.equals("hello")) {  // true
16             System.out.println("s1 equals \"hello\"");
17          }
18          else {
19             System.out.println("s1 does not equal \"hello\"");
20          }
21
```

```
s1 equals "hello"
```

For `String`s, the `equals` method determines whether the contents of the two `String`s are identical. If so, it returns `true`; otherwise, it returns `false`. Line 15's condition is `true` because `String s1` was initialized with the literal `"hello"`. Method `equals` uses a lexicographical comparison. So, if the `String "hello"` is compared to the `String "HELLO"`, the result is `false`, because the integer representation of `'h'` is different from that of `'H'`.

Comparing `String`s with the `==` Operator

Line 23's condition uses the equality operator `==` to compare `String s1` for equality with the `String` literal `"hello"`.

Click here to view code image

```
22          // test for equality with ==
23          if (s1 == "hello") { // false; they are not the same object
24             System.out.println("s1 is the same object as \"hello\"");
25          }
26          else {
27             System.out.println("s1 is not the same object as \"hello\"");
28          }
29
```

```
s1 is not the same object as "hello"
```

When comparing primitive-type values with `==`, the result is `true` if both values are identical. When comparing references with `==`, the result is `true` if both refer to the same object in memory. You must call a method to compare objects' contents for equality. For `String`s, that method is `equals`. Line 23's condition evaluates to `false` because the reference `s1` was initialized with the statement

```
s1 = new String("hello");
```

which creates a new `String` object containing a copy of `"hello"` and assigns the new object to variable `s1`. The condition would be `true` if `s1` had been initialized by directly assigning `"hello"` to variable `s1` as in:

```
s1 = "hello";
```

Remember that Java treats all `String` literals with the same contents as one `String` object with many references to it. Thus, the `"hello"` literals in lines 6, 15 and 23 all refer to the same `String` object.

`String` Method `equalsIgnoreCase`

When sorting `String`s, you might compare them for equality with the **`String` class's `equalsIgnoreCase` method**, which performs a case-insensitive comparison. Thus, `"hello"` and `"HELLO"` compare as equal. Line 31 uses `equalsIgnoreCase` to compare `String s3`—`Happy Birthday`—for equality with `String s4`—`happy birthday`. This comparison is `true` because the method ignores the case.

Click here to view code image

```
30          // test for equality (ignore case)
31          if (s3.equalsIgnoreCase(s4)) { // true
32             System.out.printf("%s equals %s with case ignored%n", s3, s4);
33          }
34          else {
35             System.out.println("s3 does not equal s4");
36          }
37
```

```
Happy Birthday equals happy birthday with case ignored
```

`String` Method `compareTo`

Lines 39–48 use the `String` **method** `compareTo` to compare `String`s. Line 40 compares `String s1` (`"hello"`) to `String s2` (`"goodbye"`). Method `compareTo` returns

- 0 if the `String`s are equal,
- a negative number if the `String` that calls the method is less than the `String` argument and
- a positive number if the `String` that calls the method is greater than the `String` argument.

The method uses lexicographical comparison to compare the Unicode numeric values of each `String`'s corresponding characters.

Click here to view code image

```
38          // test compareTo
39          System.out.printf(
40             "%ns1.compareTo(s2) is %d", s1.compareTo(s2));
41          System.out.printf(
42             "%ns2.compareTo(s1) is %d", s2.compareTo(s1));
43          System.out.printf(
44             "%ns1.compareTo(s1) is %d", s1.compareTo(s1));
45          System.out.printf(
46             "%ns3.compareTo(s4) is %d", s3.compareTo(s4));
47          System.out.printf(
48             "%ns4.compareTo(s3) is %d%n%n", s4.compareTo(s3));
49
```

```
s1.compareTo(s2) is 1
s2.compareTo(s1) is -1
s1.compareTo(s1) is 0
s3.compareTo(s4) is -32
s4.compareTo(s3) is 32
```

`String` Method `regionMatches`

The condition at line 51 uses a version of the `String` **method** `regionMatches` to compare portions of two `String`s for equality. The first argument to this version of the method is the starting index in the `String` that invokes the method. The second argument is a comparison `String`. The third argument is the starting index in the comparison `String`. The last argument is the number of characters to compare. The method returns `true` only if the characters compared are lexicographically equal.

Finally, the condition at line 60 uses a five-argument version of

`regionMatches` to compare portions of two `String`s for equality. When the first argument is `true`, the method ignores characters' case. The remaining arguments are identical to those described for the four-argument `regionMatches` method.

Click here to view code image

```
50          // test regionMatches (case sensitive)
51          if (s3.regionMatches(0, s4, 0, 5)) {
52             System.out.println("First 5 characters of s3 and s4 match");
53          }
54          else {
55             System.out.println(
56                "First 5 characters of s3 and s4 do not match");
57          }
58
59          // test regionMatches (ignore case)
60          if (s3.regionMatches(true, 0, s4, 0, 5)) {
61             System.out.println(
62                "First 5 characters of s3 and s4 match with case ignored\n");
63          }
64          else {
65             System.out.println(
66                "First 5 characters of s3 and s4 do not match\n");
67          }
68
```

```
First 5 characters of s3 and s4 do not match
First 5 characters of s3 and s4 match with case ignored
```

`String` Method `isBlank`

Line 70 uses **`String` method `isBlank`** to determine whether `" \t \n"` contains only whitespace characters. In this case, `isBlank` returns `true` because the `String` contains three whitespace characters—a space, a tab and a newline.

Click here to view code image

```
69          // test whether a String contains only whitespace
70          System.out.printf("\" \\t\\n\".isBlank(): %b%n", " \t\n".isBlank());
71       }
72    }
```

```
" \t\n".isBlank(): true
```

`String` Methods `startsWith` and `endsWith`

Figure 7.4 demonstrates `String` methods `startsWith` (lines 10 and 19) and `endsWith` (line 29). Method `main` creates the array `strings` containing `"started"`, `"starting"`, `"ended"` and `"ending"`. We use three `for` statements to test the array elements, determining whether they start with or end with a particular set of characters.

Click here to view code image

```
1  // Fig. 7.4: StringStartEnd.java
2  // String methods startsWith and endsWith.
3
4  public class StringStartEnd {
5     public static void main(String[] args) {
6        String[] strings = {"started", "starting", "ended", "ending"};
7
8        // test method startsWith
9        for (String string : strings) {
10          if (string.startsWith("st")) {
11             System.out.printf("\"%s\" starts with \"st\"%n", string);
12          }
13       }
14
15       System.out.println();
16
17       // test method startsWith starting from position 2 of string
18       for (String string : strings) {
19          if (string.startsWith("art", 2)) {
20             System.out.printf(
21                "\"%s\" starts with \"art\" at position 2%n", string);
22          }
23       }
24
25       System.out.println();
26
27       // test method endsWith
28       for (String string : strings) {
29          if (string.endsWith("ed")) {
30             System.out.printf("\"%s\" ends with \"ed\"%n", string);
31          }
32       }
33    }
34 }
```

```
"started" starts with "st"
"starting" starts with "st"

"started" starts with "art" at position 2
```

```
"starting" starts with "art" at position 2

"started" ends with "ed"
"ended" ends with "ed"
```

Fig. 7.4 | `String` methods `startsWith` and `endsWith`.

Lines 9–13 use the version of method `startsWith` that takes a `String` argument. The condition in the `if` statement (line 10) determines whether each `String` starts with the characters `"st"`. If so, the method returns `true`, and the program prints that `String`; otherwise, the method returns `false` and nothing is printed.

Lines 18–23 use the `startsWith` method that takes a `String` and an integer as arguments. The integer specifies the index in the `String` where the comparison should begin. The condition in line 19 determines whether each `String` has the characters `"art"` starting with the character at index `2` in each `String`. If so, the method returns `true`, and the program prints the `String`.

The third `for` statement (lines 28–32) uses the method `endsWith`, which takes a `String` argument. The condition in line 29 determines whether each `String` ends with the characters `"ed"`. If so, the method returns `true`, and the program prints the `String`.

Generative AI

1 Prompt genAIs to write Java code proving that all instances of a `String` literal in a program are one object in memory, whereas a new `String` object initialized with that same `String` literal is a different object. Ask the generative AIs to explain precisely how those comparisons work.
2 Prompt genAIs asking them to explain why Java's `String` method `compareTo` does not always return `-1` or `1` when `String`s are not equal.

7.3.4 Locating Characters and Substrings in `String`s

Often, it's useful to search a `String` for a character or set of characters (known as a **substring**). For example, if you're creating your own text editor, you might include a capability to search for text in the current document. Figure 7.5 demonstrates the several versions of the **`String` methods** `indexOf` and `lastIndexOf` that

search for a specified character or substring in a `String`. We broke this example into pieces for discussion purposes. This example's searches use the `String letters` (`"abcdefghijklmabcdefghijklm"`).

Click here to view code image

```
1  // Fig. 7.5: StringIndexMethods.java
2  // String searching methods indexOf and lastIndexOf.
3
4  public class StringIndexMethods {
5     public static void main(String[] args) {
6        String letters = "abcdefghijklmabcdefghijklm";
7
```

Fig. 7.5 | String-searching methods `indexOf` and `lastIndexOf`.

Searching for Characters with Method `indexOf`

Lines 9–14 use method `indexOf` to locate the first occurrence of a character in a `String`. If the method finds the character, it returns the character's index in the `String`—otherwise, it returns `-1`. There are two versions of `indexOf` that search for characters in a `String`. The expression in line 10 uses the `indexOf` version that takes an integer representation of the character to find. The expression at line 12 uses another version of `indexOf` that takes two integer arguments—the character to find and the starting index at which the search should begin.

Click here to view code image

```
8        // test indexOf to locate a character in a String
9        System.out.printf(
10          "'c' is located at index %d%n", letters.indexOf('c'));
11       System.out.printf(
12          "'a' is located at index %d%n", letters.indexOf('a', 1));
13       System.out.printf(
14          "'$' is located at index %d%n%n", letters.indexOf('$'));
15
```

```
'c' is located at index 2
'a' is located at index 13
'$' is located at index -1
```

Searching for Characters with Method `lastIndexOf`

Lines 17–22 use `lastIndexOf` to locate the last occurrence of a

character in a `String`. The method searches from the end of the `String`. If it finds the character, it returns the character's index—otherwise, it returns –1. There are two versions of `lastIndexOf`. The expression at line 18 uses the version that takes the `int` representation of the character to find. The expression at line 20 uses the version that takes two `int` arguments—the `int` representation of the character to find and the index from which to begin searching backward.

Click here to view code image

```
16         // test lastIndexOf to find a character in a String
17         System.out.printf("Last 'c' is located at index %d%n",
18            letters.lastIndexOf('c'));
19         System.out.printf("Last 'a' is located at index %d%n",
20            letters.lastIndexOf('a', 25));
21         System.out.printf("Last '$' is located at index %d%n%n",
22            letters.lastIndexOf('$'));
23
```

```
Last 'c' is located at index 15
Last 'a' is located at index 13
Last '$' is located at index -1
```

Searching for Substrings with `indexOf` and `lastIndexOf`

Lines 25–38 demonstrate versions of methods `indexOf` and `lastIndexOf` that take a `String` as the first argument. These versions perform identically to those described earlier but they search for substrings specified by their `String` arguments. If the substring is found, these methods return the index in the `String` of the first character in the substring.

Click here to view code image

```
24         // test indexOf to locate a substring in a String
25         System.out.printf("\"def\" is located at index %d%n",
26            letters.indexOf("def"));
27         System.out.printf("\"def\" is located at index %d%n",
28            letters.indexOf("def", 7));
29         System.out.printf("\"hello\" is located at index %d%n%n",
30            letters.indexOf("hello"));
31
32         // test lastIndexOf to find a substring in a String
33         System.out.printf("Last \"def\" is located at index %d%n",
```

```
34            letters.lastIndexOf("def"));
35         System.out.printf("Last \"def\" is located at index %d%n",
36            letters.lastIndexOf("def", 25));
37         System.out.printf("Last \"hello\" is located at index %d%n",
38            letters.lastIndexOf("hello"));
39      }
40   }
```

```
"def" is located at index 3
"def" is located at index 16
"hello" is located at index -1

Last "def" is located at index 16
Last "def" is located at index 16
Last "hello" is located at index -1
```

Generative AI

1 Prompt genAIs to write statements that do the following:

a. Write Java code that locates the first occurrence of the substring `"be"` in the `String "to be or not to be"`, then display the index.

b. Write a Java expression that locates the last occurrence of the substring `"be"` in the `String "to be or not to be"`, then display the index.

Compare the generated results to the following code:

Click here to view code image

```
String s = "to be or not to be";
System.out.printf("First occurrence of \"be\" is at index: %d%n",
   s.indexOf("be"));
System.out.printf("Last occurrence of \"be\" is at index: %d%n",
   s.lastIndexOf("be"));
```

```
First occurrence of "be" is at index: 3
Last occurrence of "be" is at index: 16
```

7.3.5 Extracting Substrings from `String`s

Each of the two **`String` class `substring` methods** creates and returns a new `String` object by copying part of an existing one.

Figure 7.6 demonstrates both methods.

Click here to view code image

```
1  // Fig. 7.6: SubString.java
2  // String class substring methods.
3
4  public class SubString {
5     public static void main(String[] args) {
6        String letters = "abcdefghijklmabcdefghijklm";
7
8        // test substring methods
9        System.out.printf("Substring from index 20 to end is \"%s\"%n",
10          letters.substring(20));
11       System.out.printf("%s \"%s\"%n",
12          "Substring from index 3 up to, but not including, 6 is",
13          letters.substring(3, 6));
14    }
15 }
```

```
Substring from index 20 to end is "hijklm"
Substring from index 3 up to, but not including, 6 is "def"
```

Fig. 7.6 | `String` class `substring` methods.

The expression `letters.substring(20)` at line 10 uses the `substring` method that takes one integer argument. The argument specifies the starting index in the original `String letters` from which to copy characters. The substring returned contains a copy of the characters from the starting index to the end of the `String`. Specifying an index outside the bounds of the `String` causes an `IndexOutOfBoundsException`.

Line 13 uses the `substring` method that takes two integer arguments—the starting index from which to copy characters in the original `String` and the index one beyond the last character to copy (i.e., copy up to, but not including, that index in the `String`). The substring returned contains a copy of the specified characters from the original `String`. An index outside the bounds of the `String` causes a `IndexOutOfBoundsException`.

Generative AI

1 Prompt genAIs to write code that catches the exception thrown when code attempts to access characters outside a `String`'s bounds. The code should catch and display `IndexOutOfBoundsException`s.

7.3.6 Concatenating `String`s

`String` method `concat` (Fig. 7.7) concatenates two `String` objects (like the + operator) and returns a new `String` object containing the characters from both original `String`s. The expression `s1.concat(s2)` at line 11 forms a `String` by appending the characters in `s2` to those in `s1`. Line 12 prints `s1` after the `String` concatenation to show that it was not modified—again, `String`s are immutable, so methods like `concat` return new `String`s.

Click here to view code image

```
1  // Fig. 7.7: StringConcatenation.java
2  // String method concat.
3
4  public class StringConcatenation {
5     public static void main(String[] args) {
6        String s1 = "Happy ";
7        String s2 = "Birthday";
8
9        System.out.printf("s1 = %s%ns2 = %s%n%n",s1, s2);
10       System.out.printf(
11          "Result of s1.concat(s2) = %s%n", s1.concat(s2));
12       System.out.printf("s1 after concatenation = %s%n", s1);
13    }
14 }
```

```
s1 = Happy
s2 = Birthday

Result of s1.concat(s2) = Happy Birthday
s1 after concatenation = Happy
```

Fig. 7.7 | `String` method `concat`.

7.3.7 Miscellaneous `String` Methods

Class `String` provides several methods that return `String`s or character arrays containing modified copies of an original `String`'s contents (Fig. 7.8). We broke this example into pieces for discussion purposes. Lines 6–10 create and display the `String` objects used in this example.

Click here to view code image

```
1  // Fig. 7.8: StringMiscellaneous2.java
2  // String methods replace, toLowerCase, toUpperCase, trim and toCharArray.
3
4  public class StringMiscellaneous2 {
5     public static void main(String[] args) {
6        String s1 = "hello";
7        String s2 = "GOODBYE";
8        String s3 = "   spaces   ";
9
10       System.out.printf("s1: \"%s\"%ns2: \"%s\"%ns3: \"%s\"", s1, s2, s3);
11
```

```
s1: "hello"
s2: "GOODBYE"
s3: "   spaces   "
```

Fig. 7.8 | `String` methods `replace`, `toLowerCase`, `toUpperCase`, `trim` and `toCharArray`.

Replacing Characters in a String

Line 14 uses **`String` method `replace`** to return a new `String` object in which every occurrence in `s1` of character `'l'` (lowercase el) is replaced with character `'L'`. Method `replace` leaves the original `String` unchanged. If there are no occurrences of the first argument in the `String`, `replace` returns the original `String`. An overloaded version of method `replace` enables you to replace substrings rather than individual characters.

Click here to view code image

```
12       // test method replace
13       System.out.printf(
14          "%n%nReplace 'l' with 'L' in s1: %s%n%n", s1.replace('l', 'L'));
15
```

```
Replace 'l' with 'L' in s1: heLLo
```

Changing a String's Case

Line 17 uses the **`String` method `toUpperCase`** to get a new `String` with uppercase letters replacing any lowercase letters in `s1`, leaving the original `String` unchanged. If there are no characters to convert, `toUpperCase` returns the original `String`. Line 18 uses the **`String` method `toLowerCase`** to get a new `String` object with

lowercase letters replacing any uppercase letters in `s2`, leaving the original `String` unchanged. If there are no characters in the original `String` to convert, `toLowerCase` returns the original `String`.

Click here to view code image

```
16          // test toLowerCase and toUpperCase
17          System.out.printf("s1.toUpperCase() = %s%n", s1.toUpperCase());
18          System.out.printf("s2.toLowerCase() = %s%n%n", s2.toLowerCase());
19
```

```
s1 = HELLO
s2 = goodbye
```

Removing Whitespace at the Ends of a `String`

Line 21 uses the **`String` method `strip`** to get a new `String` with all whitespace characters at the original `String`'s beginning and/or end removed. The method returns a `String` without leading or trailing whitespace and leaves the original `String` unchanged. If there are no whitespace characters at the beginning and end, `strip` returns the original `String`. You can remove whitespace characters at only the `String`'s beginning or end using **`String` methods `stripLeading` and `stripTrailing`**, respectively.

Click here to view code image

```
20          // test strip method
21          System.out.printf("s3 after strip = \"%s\"%n%n", s3.strip());
22
```

```
s3 after strip = "spaces"
```

Converting a `String` to a `char` Array

Since `String`s are immutable, you cannot modify their characters. If you need to manipulate and modify a `String`'s characters, you can use the **`String` method `toCharArray`** to create a new `char` array containing a copy of the characters (line 24), then manipulate the characters in the array. Lines 27–29 simply output each of the array's characters separated from the next by a space.

Click here to view code image

```
        // test toCharArray method
        char[] charArray = s1.toCharArray();
        System.out.print("s1 as a character array = ");

        for (char character : charArray) {
           System.out.printf("%c ", character);
        }

        System.out.println();
     }
  }
```

```
s1 as a character array = h e l l o
```

7.3.8 `String` Method `valueOf`

Every object in Java has a `toString` method that returns the object's `String` representation. Unfortunately, this cannot be used with primitive types because they do not have methods. Class `String` provides `static` methods that take an argument of any type and convert it to a `String` object. Figure 7.9 demonstrates the **`String` class `valueOf` methods**.

Click here to view code image

```
// Fig. 7.9: StringValueOf.java
// String valueOf methods.

public class StringValueOf {
   public static void main(String[] args) {
      char[] charArray = {'a', 'b', 'c', 'd', 'e', 'f'};
      boolean booleanValue = true;
      char characterValue = 'Z';
      int integerValue = 7;
      long longValue = 10000000000L; // L suffix indicates long
      float floatValue = 2.5f; // f indicates that 2.5 is a float
      double doubleValue = 33.333; // no suffix, double is default
      Object objectRef = "hello"; // assign String to an Object reference

      System.out.printf(
         "char array = %s%n", String.valueOf(charArray));
      System.out.printf("part of char array = %s%n",
         String.valueOf(charArray, 3, 3));
      System.out.printf(
         "boolean = %s%n", String.valueOf(booleanValue));
      System.out.printf(
         "char = %s%n", String.valueOf(characterValue));
```

```
23          System.out.printf("int = %s%n", String.valueOf(integerValue));
24          System.out.printf("long = %s%n", String.valueOf(longValue));
25          System.out.printf("float = %s%n", String.valueOf(floatValue));
26          System.out.printf(
27             "double = %s%n", String.valueOf(doubleValue));
28          System.out.printf("Object = %s%n", String.valueOf(objectRef));
29       }
30    }
```

```
char array = abcdef
part of char array = def
boolean = true
char = Z
int = 7
long = 10000000000
float = 2.5
double = 33.333
Object = hello
```

Fig. 7.9 | `String valueOf` methods.

Lines 10–11 use literal values `10000000000L` and `2.5f` as the initial values of the `long` variable `longValue` and the `float` variable `floatValue`, respectively. Recall that Java treats integer literals as type `int` and floating-point literals as type `double`. Appending the letter `L` to the literal `10000000000` and appending the letter `f` to the literal `2.5` tells the compiler to treat `10000000000` as a `long` and `2.5` as a `float`. You can denote a `long` value with an uppercase `L` or lowercase `l`, and can denote a `float` value with an uppercase `F` or lowercase `f`. Uppercase `L` is preferred for `long` values because lowercase `l` can be mistaken as a `1` (one).

Line 16 creates a new `String` object with a copy of the contents of `charArray`. Line 18 creates a new `String` object by copying a portion of `charArray`—the second argument specifies the starting index from which to copy characters, and the third argument specifies the number of characters to copy.

Lines 19–28 demonstrate seven other `valueOf` overloads that take arguments of type `boolean`, `char`, `int`, `long`, `float`, `double` and `Object`, respectively. Method `valueOf` with an `Object` argument simply returns the result of calling the argument's `toString` method.

Generative AI

1 Most `String.valueOf` calls are not required because Java can

convert every primitive-type value and every object to a `String` automatically. Prompt genAIs with Fig. 7.9's code and ask them to remove as many of the `String.valueOf` calls as possible, while still using `%s` to output the values. Compare the generated code to Fig. 7.9.

7.4 Class `StringBuilder`

We now discuss the **`StringBuilder` class's** features for creating and manipulating modifiable string information. Every `StringBuilder` can store a number of characters specified by its capacity. When that capacity is exceeded, a `StringBuilder` expands its capacity to accommodate more characters—the new capacity is twice the old capacity plus 2.

In programs that frequently perform `String` concatenation or other `String` modifications, implementing those operations with a `StringBuilder` is more efficient and avoids creating unnecessary extra `String` objects. Java can perform optimizations involving `String` objects—such as referring to one `String` object from multiple variables—because it knows these objects will not change. Use `String`s for text that does not change.

7.4.1 Creating `StringBuilder` Objects

Class `StringBuilder` provides four constructors for initializing new `StringBuilder` objects. Figure 7.10 demonstrates three of these:

- Line 6 uses the no-argument `StringBuilder` constructor to create an empty `StringBuilder` with a default capacity of 16 characters.
- Line 7 uses the `StringBuilder` constructor with an integer argument to create an empty `StringBuilder` and the specified initial capacity (`10` in this case). If you know the final size of the content you'll build, you can use this constructor to preallocate that capacity, making your program more efficient because the `StringBuilder` will not have to expand to accommodate more text.
- Line 8 uses the `StringBuilder` constructor that takes a `String` argument to create a `StringBuilder` containing the `String`'s characters. The `StringBuilder`'s initial capacity is the `String`'s length plus 16.

Lines 10–12 implicitly invoke `StringBuilder` method `toString` to output the contents of each `StringBuilder`.

Click here to view code image

```
1  // Fig. 7.10: StringBuilderConstructors.java
2  // StringBuilder constructors.
3
4  public class StringBuilderConstructors {
5     public static void main(String[] args) {
6        var buffer1 = new StringBuilder();
7        var buffer2 = new StringBuilder(10);
8        var buffer3 = new StringBuilder("hello");
9
10       System.out.printf("buffer1 = \"%s\"%n", buffer1);
11       System.out.printf("buffer2 = \"%s\"%n", buffer2);
12       System.out.printf("buffer3 = \"%s\"%n", buffer3);
13    }
14 }
```

```
buffer1 = ""
buffer2 = ""
buffer3 = "hello"
```

Fig. 7.10 | `StringBuilder` constructors.

7.4.2 Methods `length`, `capacity`, `setLength` and `ensureCapacity`

The **`StringBuilder` class's `length` and `capacity` methods** return the number of characters currently in a `StringBuilder` and the number of characters that can be stored without allocating more memory, respectively. **`StringBuilder`'s `ensureCapacity` method** guarantees that a `StringBuilder` has at least the specified capacity, increasing the capacity if necessary. **`StringBuilder`'s `setLength` method** increases or decreases the length of a `StringBuilder`. Figure 7.11 demonstrates these methods.

Click here to view code image

```
1  // Fig. 7.11: StringBuilderCapLen.java
2  // StringBuilder length, setLength, capacity and ensureCapacity methods.
3
4  public class StringBuilderCapLen {
5     public static void main(String[] args) {
```

```
6          var buffer = new StringBuilder("Hello, how are you?");
7
8          System.out.printf("buffer = %s%nlength = %d%ncapacity = %d%n%n",
9             buffer.toString(), buffer.length(), buffer.capacity());
10
11         buffer.ensureCapacity(75);
12         System.out.printf("New capacity = %d%n%n", buffer.capacity());
13
14         buffer.setLength(10);
15         System.out.printf("New length = %d%nbuffer = %s%n",
16            buffer.length(), buffer.toString());
17      }
18   }
```

```
buffer = Hello, how are you?
length = 19
capacity = 35

New capacity = 75

New length = 10
buffer = Hello, how
```

Fig. 7.11 | `StringBuilder` `length`, `setLength`, `capacity` and `ensureCapacity` methods.

The program contains one `StringBuilder` called `buffer`. Line 6 uses the `StringBuilder` constructor that takes a `String` argument to initialize the `StringBuilder` with `"Hello, how are you?"`. Lines 8–9 print the `StringBuilder`'s contents, length and capacity. The `buffer`'s initial capacity is 35 because the constructor that takes a `String` argument initializes the capacity to the `String`'s length plus 16.

Line 11 calls `ensureCapacity` to expand the `StringBuilder`'s capacity to a minimum of 75 characters. The `StringBuilder`'s capacity remains unchanged if it's more than method `ensureCapacity`'s argument.

Dynamically increasing a `StringBuilder`'s capacity frequently can degrade performance. If a `StringBuilder`'s size will increase repeatedly, setting its capacity high will improve performance by reducing the number of times the `StringBuilder` needs to reallocate its internal array.

Line 14 uses method `setLength` to set the length of the `StringBuilder` to 10. If the specified length is less than the `StringBuilder`'s length, its contents are truncated to the specified

length—the subsequent characters are discarded. If the specified length is greater than the `StringBuilder`'s length, `null` characters (`'\0'`) are appended until the `StringBuilder`'s total number of characters is equal to the specified length.

Generative AI

1 Prompt genAIs to explain why a `StringBuilder` adds 16 to its capacity when you initialize it with a `String`.
2 Section 7.4's introduction mentioned that when a `StringBuilder`'s capacity is exceeded, it expands such that "the new capacity is twice the old capacity plus 2." Prompt genAIs to write Java code demonstrating this growth pattern.

7.4.3 Methods `charAt`, `setCharAt`, `getChars` and `reverse`

You can manipulate a `StringBuilder`'s characters with **`StringBuilder` methods** **`charAt`**, **`setCharAt`**, **`getChars`** and **`reverse`** (Fig. 7.12):

- Method `charAt` (line 10) takes an integer argument and returns the character in the `StringBuilder` at that index.
- Method `getChars` (line 13) copies characters from a `StringBuilder` into the character array passed as an argument. This method takes four arguments—the starting index from which characters should be copied, the index one past the last character to be copied, the character array into which to copy the characters and the starting index in the character array where the first copied character should be placed.
- Method `setCharAt` (lines 20 and 21) takes an index and a character argument, and replaces the `StringBuilder`'s character at that index with the character argument.
- Method `reverse` (line 24) reverses the contents of the `StringBuilder`.

For the `StringBuilder` methods that receive indices, attempting to access a character outside the bounds of the `StringBuilder` causes a `IndexOutOfBoundsException`.

Click here to view code image

```
1  // Fig. 7.12: StringBuilderChars.java
2  // StringBuilder methods charAt, setCharAt, getChars and reverse.
3
4  public class StringBuilderChars {
5     public static void main(String[] args) {
6        var buffer = new StringBuilder("hello there");
7
8        System.out.printf("buffer = %s%n", buffer.toString());
9        System.out.printf("Character at 0: %s%nCharacter at 4: %s%n%n",
10          buffer.charAt(0), buffer.charAt(4));
11
12       char[] charArray = new char[buffer.length()];
13       buffer.getChars(0, buffer.length(), charArray, 0);
14       System.out.print("The characters are: ");
15
16       for (char character : charArray) {
17          System.out.print(character);
18       }
19
20       buffer.setCharAt(0, 'H');
21       buffer.setCharAt(6, 'T');
22       System.out.printf("%n%nbuffer = %s", buffer.toString());
23
24       buffer.reverse();
25       System.out.printf("%n%nbuffer = %s%n", buffer.toString());
26    }
27 }
```

```
buffer = hello there
Character at 0: h
Character at 4: o

The characters are: hello there

buffer = Hello There

buffer = erehT olleH
```

Fig. 7.12 | `StringBuilder` methods `charAt`, `setCharAt`, `getChars` and `reverse`.

Generative AI

1 Prompt genAIs with the code in Fig. 7.12, asking them to simplify the code and explain the changes they make.

7.4.4 `append` Methods

The **`StringBuilder` class's overloaded `append` methods** (Fig. 7.13) add values of various types to the end of a `StringBuilder`. Versions are provided for each primitive type, character arrays, `String`s, `Object`s, and more. Each method takes its argument, converts it to a `String` and appends it to the `StringBuilder`. In this example, the call

```
System.lineSeparator()
```

returns the system-specific line separator, which varies among operating systems.

Click here to view code image

```
1  // Fig. 7.13: StringBuilderAppend.java
2  // StringBuilder append methods.
3
4  public class StringBuilderAppend {
5     public static void main(String[] args) {
6        Object objectRef = "hello";
7        String string = "goodbye";
8        char[] charArray = {'a', 'b', 'c', 'd', 'e', 'f'};
9        boolean booleanValue = true;
10       char characterValue = 'Z';
11       int integerValue = 7;
12       long longValue = 10000000000L;
13       float floatValue = 2.5f;
14       double doubleValue = 33.333;
15
16       var lastBuffer = new StringBuilder("last buffer");
17       var buffer = new StringBuilder();
18
19       buffer.append(objectRef)
20             .append(System.lineSeparator())
21             .append(string)
22             .append(System.lineSeparator())
23             .append(charArray)
24             .append(System.lineSeparator())
25             .append(charArray, 0, 3)
26             .append(System.lineSeparator())
27             .append(booleanValue)
28             .append(System.lineSeparator())
29             .append(characterValue)
30             .append(System.lineSeparator())
31             .append(integerValue)
32             .append(System.lineSeparator())
```

```
                .append(longValue)
                .append(System.lineSeparator())
                .append(floatValue)
                .append(System.lineSeparator())
                .append(doubleValue)
                .append(System.lineSeparator())
                .append(lastBuffer);

        System.out.printf("buffer contains%n%s%n", buffer.toString());
    }
}
```

```
buffer contains
hello
goodbye
abcdef
abc
true
Z
7
10000000000
2.5
33.333
last buffer
```

Fig. 7.13 | `StringBuilder` `append` methods.

Implementing `+` and `+=` for Strings

The compiler can use `StringBuilder` and the `append` methods to implement the `+` and `+=` `String` concatenation operators to avoid creating unnecessary extra `String` objects. For example, assuming the declarations

```
String string1 = "hello";
String string2 = "BC";
int value = 22;
```

the statement

Click here to view code image

```
String s = string1 + string2 + value;
```

concatenates `"hello"`, `"BC"` and `22`. The concatenation can be performed as follows:

Click here to view code image

```
String s = new StringBuilder().append("hello").append("BC").
   append(22).toString();
```

First, the preceding statement creates an empty `StringBuilder`, then appends the `String`s `"hello"` and `"BC"` and the integer `22`. Next, `StringBuilder`'s `toString` method converts the `StringBuilder` to a `String` object to be assigned to `String s`. The statement

```
s += "!";
```

can be performed as follows:

Click here to view code image

```
s = new StringBuilder().append(s).append("!").toString();
```

This creates an empty `StringBuilder`, then appends the current contents of `s` followed by `"!"`. Next, `StringBuilder`'s method `toString` (which must be called explicitly here) returns the `StringBuilder`'s contents as a `String`, and the result is assigned to `s`.

Generative AI

1 Prompt genAIs to rewrite the code in Fig. 7.13 to use `String` concatenation with the `+` operator, then compare the generated code to Fig. 7.13.

7.4.5 Insertion and Deletion Methods

The **`StringBuilder` class's overloaded `insert` methods** insert values of various types at any valid position in a `StringBuilder`. Versions are provided for the primitive types, character arrays, `String`s, `Object`s and more. Each method takes its second argument and inserts it at the index specified by the first argument. If the first argument is less than `0` or greater than the `StringBuilder`'s length, a `StringIndexOutOfBoundsException` occurs. The **`StringBuilder` class's `delete` and `deleteCharAt` methods** delete characters at any position in a `StringBuilder`:

- `delete` takes two arguments—the starting index and the index one past the end of the characters to delete. All characters beginning at the starting index up to but not

including the ending index are deleted.

- `deleteCharAt` takes one argument—the index of the character to delete.

Invalid indices cause both methods to throw a `StringIndexOutOfBoundsException`. Figure 7.14 demonstrates methods `insert`, `delete` and `deleteCharAt`.

Click here to view code image

```
// Fig. 7.14: StringBuilderInsertDelete.java
// StringBuilder methods insert, delete and deleteCharAt.

public class StringBuilderInsertDelete {
   public static void main(String[] args) {
      Object objectRef = "hello";
      String string = "goodbye";
      char[] charArray = {'a', 'b', 'c', 'd', 'e', 'f'};
      boolean booleanValue = true;
      char characterValue = 'K';
      int integerValue = 7;
      long longValue = 10000000;
      float floatValue = 2.5f; // f suffix indicates that 2.5 is a float
      double doubleValue = 33.333;

      var buffer = new StringBuilder();

      buffer.insert(0, objectRef);
      buffer.insert(0, "  "); // each of these contains two spaces
      buffer.insert(0, string);
      buffer.insert(0, "  ");
      buffer.insert(0, charArray);
      buffer.insert(0, "  ");
      buffer.insert(0, charArray, 3, 3);
      buffer.insert(0, "  ");
      buffer.insert(0, booleanValue);
      buffer.insert(0, "  ");
      buffer.insert(0, characterValue);
      buffer.insert(0, "  ");
      buffer.insert(0, integerValue);
      buffer.insert(0, "  ");
      buffer.insert(0, longValue);
      buffer.insert(0, "  ");
      buffer.insert(0, floatValue);
      buffer.insert(0, "  ");
      buffer.insert(0, doubleValue);

      System.out.printf(
         "buffer after inserts:%n%s%n%n", buffer.toString());
```

```
40
41          buffer.deleteCharAt(10); // delete 5 in 2.5
42          buffer.delete(2, 6); // delete .333 in 33.333
43
44          System.out.printf(
45             "buffer after deletes:%n%s%n", buffer.toString());
46       }
47    }
```

```
buffer after inserts:
33.333  2.5  10000000  7  K  true  def  abcdef  goodbye  hello

buffer after deletes:
33  2.  10000000  7  K  true  def  abcdef  goodbye  hello
```

Fig. 7.14 | `StringBuilder` methods `insert`, `delete` and `deleteCharAt`.

Generative AI

1 Prompt genAIs asking why some Java `String` and `StringBuilder` methods throw `IndexOutOfBoundsExceptions` but others throw `StringIndexOutOfBoundsException`s?

7.5 Class `Character`

Java provides eight **type-wrapper classes** that treat primitive-type values as objects—`Boolean`, `Character`, `Double`, `Float`, `Byte`, `Short`, `Integer` and `Long`. This section presents class `Character`—the type-wrapper class for primitive type `char`.

Class `Character`[4] provides `static` methods for conveniently processing individual `char` values. These methods generally receive a `char` argument and either test it or manipulate it. For more information on class `Character` (and all the type-wrapper classes), refer to the `java.lang` package in the Java API documentation.

4. "Class `Character`," Accessed September 21, 2024. https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/lang/Character.html.

7.5.1 `Character` Testing and Case Conversion Methods

Figure 7.15 demonstrates

- `static Character` methods that test characters to determine whether they're a specific character type,
- `static Character` methods that perform case conversions on characters.

Many of the `Character` testing methods shown here (and the techniques in Section 7.8) are useful for validating program inputs. Lines 8–10 input a character to process. **Scanner's `next` method** (line 9) reads characters until it encounters a whitespace character, then returns a `String` containing the characters up to, but not including, the whitespace character, which is discarded.

Click here to view code image

```
1  // Fig. 7.15: StaticCharMethods.java
2  // Character static methods for testing characters and converting case.
3  import java.util.Scanner;
4
5  public class StaticCharMethods {
6     public static void main(String[] args) {
7        var scanner = new Scanner(System.in); // create scanner
8        System.out.println("Enter a character and press Enter");
9        String input = scanner.next();
10       char c = input.charAt(0); // get input character
11
12       // display character info
13       System.out.printf("is defined: %b%n", Character.isDefined(c));
14       System.out.printf("is digit: %b%n", Character.isDigit(c));
15       System.out.printf("is first character in a Java identifier: %b%n",
16          Character.isJavaIdentifierStart(c));
17       System.out.printf("is part of a Java identifier: %b%n",
18          Character.isJavaIdentifierPart(c));
19       System.out.printf("is letter: %b%n", Character.isLetter(c));
20       System.out.printf(
21          "is letter or digit: %b%n", Character.isLetterOrDigit(c));
22       System.out.printf(
23          "is lower case: %b%n", Character.isLowerCase(c));
24       System.out.printf(
25          "is upper case: %b%n", Character.isUpperCase(c));
26       System.out.printf(
27          "to upper case: %c%n", Character.toUpperCase(c));
28       System.out.printf(
29          "to lower case: %c%n", Character.toLowerCase(c));
30    }
31 }
```

```
Enter a character and press Enter
A
is defined: true
is digit: false
is first character in a Java identifier: true
is part of a Java identifier: true
is letter: true
is letter or digit: true
is lower case: false
is upper case: true
to upper case: A
to lower case: a
```

```
Enter a character and press Enter
8
is defined: true
is digit: true
is first character in a Java identifier: false
is part of a Java identifier: true
is letter: false
is letter or digit: true
is lower case: false
is upper case: false
to upper case: 8
to lower case: 8
```

```
Enter a character and press Enter
$
is defined: true
is digit: false
is first character in a Java identifier: true
is part of a Java identifier: true
is letter: false
is letter or digit: false
is lower case: false
is upper case: false
to upper case: $
to lower case: $
```

Fig. 7.15 | `Character` `static` methods for testing characters and converting case.

Each `Character` method that begins with the word `"is"` returns `true` or `false`:

- Line 13 uses the **`Character` method `isDefined`** to determine whether character `c` is defined in the Unicode character set.

- Line 14 uses the `Character` **method** `isDigit` to determine whether character `c` is a defined Unicode digit.
- Line 16 uses the `Character` **method** `isJavaIdentifierStart` to determine whether `c` is a character that can be the first character of a Java identifier—a letter, an underscore (_) or a dollar sign (`$`).
- Line 18 uses the `Character` **method** `isJavaIdentifierPart` to determine whether character `c` is a character that can be used in an identifier in Java—that is, a digit, a letter, an underscore (_) or a dollar sign (`$`).
- Line 19 uses the `Character` **method** `isLetter` to determine whether character `c` is a letter.
- Line 21 uses `Character` method `isLetterOrDigit` to determine whether character `c` is a letter or a digit.
- Line 23 uses the `Character` **method** `isLowerCase` to determine whether character `c` is a lowercase letter.
- Line 25 uses the `Character` **method** `isUpperCase` to determine whether character `c` is an uppercase letter.
- Line 27 uses the `Character` **method** `toUpperCase` to convert `c` to its uppercase equivalent. The method returns the converted character if the character has an uppercase equivalent; otherwise, the method returns its argument.
- Line 29 uses the `Character` **method** `toLowerCase` to convert `c` to its lowercase equivalent. The method returns the converted character if the character has a lowercase equivalent; otherwise, the method returns its argument.

7.5.2 Character/Digit Conversions

Figure 7.16 demonstrates `Character` **methods** `digit` **and** `forDigit`, which convert characters to digits and vice versa in different number systems—decimal (base 10), octal (base 8), hexadecimal (base 16) and binary (base 2). The base of a number is also known as its **radix**. For more information on conversions between number systems, refer to Appendix C.

Click here to view code image

```
1  // Fig. 7.16: StaticCharMethods2.java
2  // Character class static conversion methods.
```

```
3   import java.util.Scanner;
4
5   public class StaticCharMethods2 {
6      public static void main(String[] args) {
7         var scanner = new Scanner(System.in);
8
9         // get radix
10        System.out.print("Please enter a radix: ");
11        int radix = scanner.nextInt();
12
13        // get user choice
14        System.out.println("""
15           Please choose one:
16              1 -- Convert digit to character
17              2 -- Convert character to digit""");
18        System.out.print(": ");
19        int choice = scanner.nextInt();
20
21        // process request
22        switch (choice) {
23           case 1 -> { // convert digit to character
24              System.out.print("Enter a digit: ");
25              int digit = scanner.nextInt();
26              System.out.printf("Convert digit to character: %s%n",
27                 Character.forDigit(digit, radix));
28           }
29           case 2 -> { // convert character to digit
30              System.out.print("Enter a character: ");
31              char character = scanner.next().charAt(0);
32              System.out.printf("Convert character to digit: %s%n",
33                 Character.digit(character, radix));
34           }
35        };
36     }
37  }
```

```
Please enter a radix: 16
Please choose one:
   1 -- Convert digit to character
   2 -- Convert character to digit
: 2
Enter a character: A
Convert character to digit: 10
```

```
Please enter a radix: 16
Please choose one:
   1 -- Convert digit to character
   2 -- Convert character to digit
```

```
: 1
Enter a digit: 13
Convert digit to character: d
```

Fig. 7.16 | `Character` class `static` conversion methods.

Line 27 uses the method `forDigit` to convert the integer `digit` into a character in the number system specified by the integer `radix` (the number's base). For example, the decimal integer `13` in base 16 (the `radix`) has the character value `'d'`. Lowercase and uppercase letters represent the same value in number systems. Line 33 uses the method `digit` to convert variable `character` into an integer in the number system specified by the integer `radix` (the base of the number). For example, the character `'A'` is the base 16 (the `radix`) representation of the base 10 value 10. The radix must be between 2 and 36, inclusive.

7.5.3 Other `Character` Methods

Figure 7.17 demonstrates the **`Character` class instance methods**—**`charValue`**, `toString` and `equals`. Lines 5–6 instantiate two `Character` objects by assigning the character constants `'A'` and `'a'`, respectively, to the `Character` variables. Java autoboxes these `char`s into `Character` objects. Line 9 uses the `Character` method `charValue` to get the `char` stored in `c1` and uses method `toString` to get a `String` representation of `c2`. The condition in line 11 uses method `equals` to determine whether the `char`s in the objects `c1` and `c2` are equal. You'd generally work with `Character` objects and their instance methods only when storing `char`s in a collection, such as an `ArrayList<Character>`. Given Java's ability to box `char`s into `Character` objects and unbox `char`s from `Character` objects, the `Character` instance methods are rarely used.

Click here to view code image

```
1  // Fig. 7.17: OtherCharMethods.java
2  // Character class instance methods.
3  public class OtherCharMethods {
4     public static void main(String[] args) {
5        Character c1 = 'A';
6        Character c2 = 'a';
7
8        System.out.printf(
9           "c1 = %c%nc2 = %s%n%n", c1.charValue(), c2.toString());
10
```

```
11          if (c1.equals(c2)) {
12             System.out.println("c1 and c2 are equal");
13          }
14          else {
15             System.out.println("c1 and c2 are not equal");
16          }
17       }
18    }
```

```
c1 = A
c2 = a

c1 and c2 are not equal
```

Fig. 7.17 | `Character` class instance methods.

7.6 Tokenizing `String`s

When you read a sentence, your mind breaks it into **tokens**—individual words and punctuation marks that convey meaning to you. Compilers also perform tokenization. They break up statements into individual pieces like keywords, identifiers, operators and other programming-language elements. In natural language processing (NLP) and generative AI, tokenizing text into sentences and individual words is a key aspect of preparing text for analysis (such as word frequency counting) and training the large language models (LLMs) that help computers understand natural language. In this section, we'll use the **`String`'s class's `split` method** to break a `String` into its component tokens. Tokens are separated from one another by **delimiters**, typically whitespace characters such as space, tab, newline and carriage return. Other characters can also be used as delimiters to separate tokens, such as tabs, commas and more. Figure 7.18 demonstrates `String`'s `split` method.

Click here to view code image

```
1  // Fig. 7.18: TokenTest.java
2  // Tokenizing with String method split
3  import java.util.Scanner;
4
5  public class TokenTest {
6     public static void main(String[] args) {
7        // get sentence
8        var scanner = new Scanner(System.in);
9        System.out.println("Enter a sentence and press Enter");
```

```
10          String sentence = scanner.nextLine();
11
12          // process user sentence
13          String[] tokens = sentence.split(" ");
14          System.out.printf("%nNumber of elements: %d%nThe tokens are:%n",
15             tokens.length);
16
17          for (String token : tokens) {
18             System.out.printf("  %s%n", token);
19          }
20       }
21    }
```

```
Enter a sentence and press Enter
This sentence has five tokens

Number of elements: 5
The tokens are:
  This
  sentence
  has
  five
  tokens
```

Fig. 7.18 | Tokenizing with `String` method `split`.

The program stores the user input in the variable `sentence`. Line 13 invokes the `String` method `split` with the `String` argument `" "`, which returns an array of `String`s. The space character in the argument `String` is the delimiter `split` uses to separate the tokens. Section 7.8 introduces pattern matching in text using regular expressions—`split`'s argument can be a regular-expression pattern for more complex tokenizing. Lines 14–15 display the `tokens` array's length, which is the number of tokens in `sentence`. Lines 17–19 output each token on a separate line.

7.7 Intro to Natural Language Processing (NLP)—at the Root of Generative AI[5]

5. Much of this intro to natural language processing is borrowed from Chapter 12, Natural Language Processing, of our Pearson textbook *Intro to Python for Computer Science and Data Science: Learning to Program with AI, Big Data and the Cloud* (https://deitel.com/pycds).

Every day, we use **natural language** in various forms of communication. Many of the following examples may be familiar to

you:

- You read your text messages and check the latest news clips.
- You have conversations with family, friends and colleagues.
- You are Deaf or hard-of-hearing and communicate with friends and family via sign language and enjoy close-captioned video programs.
- You are blind or have low vision, and read braille, listen to audiobooks, and listen to a screen reader speak about what's on the computer screen.
- You read e-mails, distinguishing junk from important communications.
- You receive a client's e-mail in a language you don't know well, translate it online, then respond in English, knowing your client can usually translate your e-mail back to their language.
- You travel through a neighborhood (maybe via car, bike, or wheelchair, or on foot), observing road signs such as "Stop" and "Pedestrian Crossing."
- You give your phone verbal commands like "call home" or "play classical music" or ask questions like, "Where's the nearest gas station?"
- You teach a child how to speak and read.
- You learn a foreign language.

Natural Language Processing (NLP) helps computers understand, analyze and process human text and speech. It's performed on text collections composed of emails, social media posts, conversations, movie reviews, Shakespeare's plays, historical documents, news items, meeting logs, and more. A text collection is known as a **corpus**, the plural of which is **corpora**.

Some typical NLP applications include:

- **Natural language understanding—**understanding text or spoken language.
- **Sentiment analysis**—determining whether text has positive, neutral or negative sentiment. For example, companies analyze the sentiment of social media posts about their products.

- **Readability assessment**—determining how readable text is based on vocabulary, word lengths, sentence lengths, sentence structure, topics covered and more. While writing this book, we used the paid versions of the NLP tool Grammarly[6] to help us tune our writing to ensure its quality and readability for a broad audience.

  6. Grammarly also has a free version (https://www.grammarly.com).

- **Intelligent virtual assistants—**software that helps you perform everyday tasks. Popular intelligent virtual assistants include Amazon Alexa, Apple Siri, Microsoft Copilot, Google Assistant, OpenAI ChatGPT and Anthropic Claude.
- **Text summarization**—summarizing the key points of a text corpus. This can save time for busy people.
- **Speech recognition**—converting speech to text.
- **Speech synthesis**—converting text to speech.
- **Language identification**—receiving a text when you don't know its language in advance, then automatically determining the language.
- **Interlanguage translation**—converting text to other spoken languages.
- **Named-entity recognition (NER)**—locating and categorizing items in a text corpus like dates, times, quantities, places, people, things, organizations and more.
- **Chatbots**—AI-based software that humans interact with via natural language. One popular chatbot application is automated customer support.
- **Similarity detection**—examining documents to determine how alike they are. Basic similarity metrics include average sentence length, frequency distribution of sentence lengths, average word length, frequency distribution of word lengths, frequency distribution of word usage, and more.

Many lower-level NLP tasks support the above applications as they perform their tasks, including:

- **Tokenization**—splitting text into tokens, which are meaningful units, such as words and numbers.
- **Parts-of-speech (POS) tagging**—identifying each word's

part of speech, such as noun, verb, adjective, etc.

- **Noun phrase extraction**—locating groups of words representing nouns, such as "the red brick factory."[7]

  7. The phrase "red brick factory" illustrates why natural language is such a challenging subject. Is a "red brick factory" a factory that makes red bricks? Is it a red factory that makes bricks of any color? Is it a factory built of red bricks that makes products of any type? In today's music world, it could even be a rock band's name or the name of a game on your smartphone.

- **Spelling and grammar checking** and **correction**.
- **Stemming**—reducing words to their stems by removing prefixes or suffixes. For example, the stem of "varieties" is "varieti."
- **Lemmatization**—like stemming, but produces actual words based on the original words' context. For example, the lemmatized form of "varieties" is "variety."
- **Word frequency counting**—determining how often each word appears in a corpus.
- **Stop-word elimination**—removing words, such as—*a*, *and*, *as*, *at*, *be*, *do*, *for*, *have*, *he*, *I*, *in*, *it*, *not*, *of*, *on*, *that*, *the*, *to*, *with*, *you* and more—to analyze the important words in a corpus.
- **n-grams**—producing sets of consecutive words in a corpus to identify words frequently appearing adjacent to one another. N-grams are commonly used for predictive text input, such as when your smartphone suggests possible next words as you enter a text message, or your coding support tool lists the possible methods for an object you're working with.

Generative AI

1 Prompt genAIs for a list of the top-20 n-grams in your preferred spoken language.

7.8 Objects-Natural Case Study: Intro to Regular Expressions in NLP

Sometimes, you'll need to recognize patterns in text, like phone numbers, e-mail addresses, ZIP codes, web addresses, Social Security numbers and more. A **regular expression (regex)** `String` describes a search pattern for matching characters in other `String`s. Regular expressions can help you extract data from unstructured text, such as social media posts. They're also crucial for preparing text data for processing so it can be used to train artificial intelligence models used in natural language processing (NLP) and generative AI. Before working with text data, you'll often use regular expressions to validate it. For example, you might check that:

- A U.S. ZIP code consists of five digits (such as 02215) or five digits followed by a hyphen and four more digits (such as 02215-4775).
- A last name contains only letters, spaces, apostrophes and hyphens.
- An e-mail address contains only the allowed characters in the allowed order.
- A U.S. Social Security number contains three digits, a hyphen, two digits, a hyphen and four digits, and adheres to other rules about the specific numbers used in each group of digits.

For common patterns, you rarely need to create regular expressions. Sites like:

- `https://regex101.com`
- `https://regexr.com`
- `https://regexlib.com`
- `https://regular-expressions.info`

offer repositories of existing regular expressions you can copy and use. These also allow you to test regular expressions to determine whether they meet your needs.

Other Uses of Regular Expressions

Regular expressions are also used to:

- Extract data from text (known as scraping, text mining and parsing)—for example, locating all URLs in a webpage.

- Clean data—for example, removing data that's not required, eliminating duplicate data, handling incomplete data, fixing typos, ensuring consistent data formats, removing formatting, changing text cases, dealing with outliers and more.
- Transform data into other formats—for example, reformatting tab-separated or space-separated values into comma-separated values (CSV) for an application that requires data to be in CSV format.

Generative AI

1 Prompt genAIs to locate the best sites for testing regular expressions. Test that the regular expressions we show in this chapter match their intended text patterns.

7.8.1 Matching Complete `Strings` to Patterns

Figure 7.19 demonstrates matching entire `String`s to regular expression patterns. We broke the program into parts for discussion purposes.

Matching Literal Characters

The **`String` class's `matches` method** returns `true` if the `String` that calls the method matches the pattern in its argument. By default, pattern matching is case-sensitive—you'll see how to perform case-insensitive matches later. Let's begin by matching literal characters that match themselves. Line 8 calls the `matches` method twice, attempting to match the pattern `"02215"` containing only literal characters (in this case, digits) that match themselves in the specified order:

- The call `"02215".matches("02215")` returns `true`—`"02215"` matches the pattern in the `matches` method's argument.
- The call `"51220".matches("02215")` returns `false`—`"51220"` and the pattern in the argument have the same digits but `"51220"` has them in the wrong order.

Click here to view code image

```
1  // Fig. 7.19: RegexExamples.java
2  // Matching entire Strings to regular expressions.
```

```
3  public class RegexExamples {
4     public static void main(String[] args) {
5        // fully match a pattern of literal characters
6        System.out.println("Matching against: 02215");
7        System.out.printf("02215: %b; 51220: %b%n%n",
8           "02215".matches("02215"), "51220".matches("02215"));
9
```

```
Matching against: 02215
02215: true; 51220: false
```

Fig. 7.19 | Matching entire `String`s to regular expressions.

Metacharacters, Character Classes and Quantifiers

Regular expressions typically contain various special symbols called **metacharacters**:

Click here to view code image

```
[]  {}  ()  \  *  +  ^  $  ?  .  |
```

The **\ metacharacter** begins each of the predefined **character classes**, several of which are shown in the following table with the groups of characters they match:

Matcher class	
\d	Any digit (0–9).
\D	Any character that is not a digit.
\s	Any whitespace character (such as a space, tab or newline).
\S	Any character that is not a whitespace character.
\w	Any **word character** (also called an **alphanumeric character**)—that is, any uppercase or lowercase letter, any digit or an underscore
\W	Any character that is not a word character.

Recall that a backslash (\) in a Java `String` begins an escape sequence. Regular expression character classes are not valid Java escape sequences, so **you must precede each character class with an extra backslash in a regular expression**. For example, `\$` matches a dollar sign (`$`) and `\\` matches a backslash (\). To represent these patterns in Java, you must **escape each backslash**, so

- to match a dollar sign (`$`), specify `\\$` in the pattern `String`,

and

- to match a backslash (\), specify `\\\\` in the pattern `String`.

Matching Digits

Let's validate a five-digit United States ZIP code. In the regular expression `\d{5}`, the character class **`\d`** is a **regular expression escape sequence** that matches one character, specifically one digit (0–9). To match more than one, follow the character class with a **quantifier**. The quantifier `{5}` repeats `\d` five times to match five consecutive digits (line 13):

- The first call to `matches` returns `true`—`"02215"` contains five consecutive digits.
- The second call to `matches` returns `false`—`"9876"` contains only four consecutive digits.

Click here to view code image

```
10          // fully match five digits
11          System.out.println("Matching against: \\d{5}");
12          System.out.printf("02215: %b; 9876: %b%n%n",
13             "02215".matches("\\d{5}"), "9876".matches("\\d{5}"));
14
```

```
Matching against: \d{5}
02215: true; 9876: false
```

Custom Character Classes

Characters in square brackets, `[]`, define a **custom character class** that matches a single character. For example, `[aeiou]` matches a lowercase vowel, `[A-Z]` matches an uppercase letter, `[a-z]` matches a lowercase letter, and `[a-zA-Z]` matches any lowercase or uppercase letter. Line 18 defines a custom character class to validate a simple first name with no spaces or punctuation. A first name might contain many letters.

- `[A-Z]` matches one uppercase letter, and
- `[a-z]*` matches any number of lowercase letters.

The **`*` quantifier** matches zero or more occurrences of the subexpression to its left (in this case, `[a-z]`). So `[A-Z][a-z]*`

matches an uppercase letter followed by any number of lowercase letters, as in `"Vivaan"`, `"Jo"` and `"E"`:

Click here to view code image

```
15          // match a word that starts with a capital letter
16          System.out.println("Matching against: [A-Z][a-z]*");
17          System.out.printf("Angel: %b; tina: %b%n%n",
18             "Angel".matches("[A-Z][a-z]*"), "tina".matches("[A-Z][a-z]*"));
19
```

```
Matching against: [A-Z][a-z]*
Angel: true; tina: false
```

When a custom character class starts with a **caret (^)**, the class matches any character that's not specified. So `[^a-z]` (line 23) matches any character that's not a lowercase letter:

Click here to view code image

```
20          // match any character that's not a lowercase letter
21          System.out.println("Matching against: [^a-z]");
22          System.out.printf("A: %b; a: %b%n%n",
23             "A".matches("[^a-z]"), "a".matches("[^a-z]"));
24
```

```
Matching against: [^a-z]
A: true; a: false
```

Metacharacters in a custom character class are treated as literal characters. So `[*+$]` (line 28) matches one `*`, `+` or `$` character:

Click here to view code image

```
25          // match metacharacters as literals in a custom character class
26          System.out.println("Matching against: [*+$]");
27          System.out.printf("*: %b; !: %b%n%n",
28             "*".matches("[*+$]"), "!".matches("[*+$]"));
29
```

```
Matching against: [*+$]
*: true; !: false
```

* vs. + Quantifier

To require at least one lowercase letter in a first name, replace the `*` quantifier in line 18 with the **`+` quantifier** (line 33), which matches at least one occurrence of the subexpression to its left (`[a-z]`):

Click here to view code image

```
30          // match a capital letter followed by at least one lowercase letter
31          System.out.println("Matching against: [A-Z][a-z]+");
32          System.out.printf("Angel: %b; T: %b%n%n",
33             "Angel".matches("[A-Z][a-z]+"), "T".matches("[A-Z][a-z]+"));
34
```

```
Matching against: [A-Z][a-z]+
Angel: true; T: false
```

Both `*` and `+` are **greedy**—they match as many characters as possible. So, the regular expression `[A-Z][a-z]+` matches any word that begins with a capital letter followed by at least one lowercase letter. You'll explore **lazy quantifiers** in a genAI exercise at the end of this section.

Other Quantifiers

The **`?` quantifier** matches zero or one occurrence of the subexpression to its left. In the regular expression `labell?ed` (line 38), the subexpression is the literal character `"l"`. So, in the `matches` calls in line 38, the regular expression matches `labelled` (the U.K. English spelling) and `labeled` (the U.S. English spelling), but in the `matches` call in line 39, the regular expression does not match the misspelled word `labellled`:

Click here to view code image

```
35          // match zero or one occurrence of a subexpression
36          System.out.println("Matching against: labell?ed");
37          System.out.printf("labelled: %b; labeled: %b; labellled: %b%n%n",
38             "labelled".matches("labell?ed"), "labeled".matches("labell?ed"),
39             "labellled".matches("labell?ed"));
40
```

```
Matching against: labell?ed
labelled: true; labeled: true; labellled: false
```

You can use the **`{n,}` quantifier** to match at least *n* occurrences

of a subexpression to its left. The regular expression in lines 44–45 (`\d{3,}`) matches `String`s containing at least three digits—recall that the `\` in this regular expression must be escaped with another `\`:

Click here to view code image

```
41          // match n (3) or more occurrences of a subexpression
42          System.out.println("Matching against: \\d{3,}");
43          System.out.printf("123: %b; 1234567890: %b; 12: %b%n%n",
44             "123".matches("\\d{3,}"), "1234567890".matches("\\d{3,}"),
45             "12".matches("\\d{3,}"));
46
```

```
Matching against: \d{3,}
123: true; 1234567890: true; 12: false
```

You can use the **`{n,m}` quantifier** to match between *n* and *m* (inclusive) occurrences of a subexpression. The regular expressions in lines 50–51 (`\d{3,6}`) match `String`s containing 3 to 6 digits:

Click here to view code image

```
47          // match n to m inclusive (3-6), occurrences of a subexpression
48          System.out.println("Matching against: \\d{3,6}");
49          System.out.printf("123: %b; 123456: %b; 1234567: %b; 12: %b%n",
50             "123".matches("\\d{3,6}"), "123456".matches("\\d{3,6}"),
51             "1234567".matches("\\d{3,6}"), "12".matches("\\d{3,6}"));
52       }
53    }
```

```
Matching against: \d{3,6}
123: true; 123456: true; 1234567: false; 12: false
```

Generative AI

1 Prompt genAIs with the regular expression `[b-df-hj-np-tv-z]`. Ask them to write Java code showing that the pattern matches only lowercase consonants. Run the code to confirm that the pattern works properly.

2 Regular expression quantifiers are greedy by default. Prompt genAIs for a brief Java tutorial with simple examples showing how lazy regular expression quantifiers match patterns.

7.8.2 Replacing Substrings

The **`String` class's `replaceAll` method** replaces patterns in a `String`. Let's convert a tab-delimited `String` to comma-delimited (Fig. 7.20). Method `replaceAll` (line 9) receives:

- the `regex` pattern to match (the tab character `"\t"`) and
- the replacement text (`","`),

and returns a new `String` containing the modifications. The `String` class also provides the **`replaceFirst` method**, which replaces only the first matching substring.

Click here to view code image

```
1  // Fig. 7.20: RegexReplacement.java
2  // Regular expression replacements.
3  public class RegexReplacement {
4     public static void main(String[] args) {
5        // replace tabs with commas
6        String s1 = "1\t2\t3\t4";
7        System.out.printf("Original string: %s%n", s1);
8        System.out.printf("New string with commas replacing tabs: %s%n",
9           s1.replaceAll("\\t", ","));
10    }
11 }
```

```
Original string: 1      2       3       4
New string with commas replacing tabs: 1,2,3,4
```

Fig. 7.20 | Regular expression replacements.

7.8.3 Searching for Matches with Classes `Pattern` and `Matcher`

In addition to class `String`'s regular-expression capabilities, the **`java.util.regex` package** provides capabilities that help developers work with regular expressions:

- An object of the **`Pattern` class** represents a regular expression.
- An object of the **`Matcher` class** contains a regular-expression pattern and a `String` in which to search for the pattern.

These classes provide more control over the regular-expression, pattern-matching process.

Finding a Match Anywhere in a `String`

To introduce `Pattern` and `Matcher`, let's search for literal substrings anywhere in the `String s1` (Fig. 7.21, line 9). Line 13 invokes the **`Pattern` class's `static` method `compile`**, which creates a `Pattern` object representing a regular expression—in this case, consisting of the literal characters `"Programming"`. The `compile` method makes regular expressions more efficient, which improves performance for regular expressions that will be used more than once—in a loop, for example, which we'll do momentarily. Line 14 calls the **`Pattern` object's `matcher` method**, which creates and returns a `Matcher` object that can search for the `Pattern`'s regular expression in the `String s1` (`"Programming is fun"`). Line 15 calls the **`Matcher` object's `find` method** to search for `"Programming"` in `s1`—the method returns `true` if the pattern is found; otherwise, it returns `false`. Lines 17–19 and 21–23 repeat the preceding process to search for the patterns `"fun"` and `"fn"`, respectively.

Click here to view code image

```
1  // Fig. 7.21: RegexMatching.java
2  // Classes Pattern and Matcher.
3  import java.util.regex.Pattern;
4  import java.util.regex.Matcher;
5
6  public class RegexMatching {
7     public static void main(String[] args) {
8        // performing a simple match
9        String s1 = "Programming is fun";
10       System.out.printf("s1: %s%n%n", s1);
11       System.out.println("Search anywhere in s1:");
12
13       Pattern pattern1 = Pattern.compile("Programming");
14       Matcher matcher1 = pattern1.matcher(s1);
15       boolean found1 = matcher1.find();
16
17       Pattern pattern2 = Pattern.compile("fun");
18       Matcher matcher2 = pattern2.matcher(s1);
19       boolean found2 = matcher2.find();
20
21       Pattern pattern3 = Pattern.compile("fn");
22       Matcher matcher3 = pattern3.matcher(s1);
23       boolean found3 = matcher3.find();
24
```

```
25          System.out.printf("Programming: %b; fun: %b; fn: %b%n%n",
26             found1, found2, found3);
27
```

```
s1: Programming is fun

Search anywhere in s1:
Programming: true; fun: true; fn: false
```

Fig. 7.21 | Classes `Pattern` and `Matcher`.

Ignoring Case in a Regular Expression and Viewing the Matching Text

Like `String` comparisons, regular expression pattern matching is case-sensitive by default. The `Pattern` class provides constants that customize how regular expressions perform matches. Lines 33–34 configure a `Pattern` that performs case-insensitive matches—as indicated by the constant **`Pattern.CASE_INSENSITIVE`**—for the literal `"Gabriela"`. Line 35 creates a `Matcher` object to search for that pattern in `s2`. Line 37 calls `Matcher` method `find` to determine whether `s2` contains the pattern. If so, the **`Matcher` method `group`** (line 39) returns the substring of `s2` that matched the pattern (`"GABRIELA"` in this case).

Click here to view code image

```
28          // ignoring case
29          String s2 = "GABRIELA ALVAREZ";
30          System.out.printf("s2: %s%n%n", s2);
31          System.out.println("Case insensitive search for Gabriela in s2:");
32
33          Pattern pattern4 =
34             Pattern.compile("Gabriela", Pattern.CASE_INSENSITIVE);
35          Matcher matcher4 = pattern4.matcher(s2);
36
37          if (matcher4.find()) {
38             System.out.printf("Gabriela found%n");
39             System.out.printf("Matched text: %s%n%n", matcher4.group());
40          }
41          else {
42             System.out.printf("Gabriela not found%n");
43          }
44
```

```
s2: GABRIELA ALVAREZ
```

```
Case insensitive search for Gabriela in s2:
Gabriela found
Matched text: GABRIELA
```

Finding All Matches in a `String`

Let's extract all the 10-digit phone numbers of the form `###-###-####` from a `String`. The following code finds each substring in `contact` (lines 46–47) matching `phonePattern` (line 48) and displays the matching text. The pattern

```
\d{3}-\d{3}-\d{4}
```

matches any substring containing three digits (`\d{3}`), a hyphen (`-`), three digits (`\d{3}`), a hyphen (`-`) and four digits (`\d{4}`)—`contact` contains two such phone numbers. Line 49 creates a `Matcher` object for locating `phonePattern` matches in `contact`. When searching for multiple matches, you call the `Matcher` method `find` in a loop condition (line 52). Lines 52–54 iterate while `find` returns `true`—that is, until there are no more matches in `contact`. If there's a match, line 53 calls the `Matcher` method `group` to get the substring that was found then displays it. The next call to `Matcher` method `find` automatically continues from the character following the preceding match.

Click here to view code image

```
45          // finding all matches
46          String contact =
47             "Lea Dubois, Home: 555-555-1234, Work: 555-555-4321";
48          Pattern phonePattern = Pattern.compile("\\d{3}-\\d{3}-\\d{4}");
49          Matcher phoneMatcher = phonePattern.matcher(contact);
50
51          System.out.printf("Finding phone numbers in:%n%s%n", contact);
52          while (phoneMatcher.find()) {
53             System.out.printf("   %s%n", phoneMatcher.group());
54          }
55       }
56    }
```

```
Finding phone numbers in:
Lea Dubois, Home: 555-555-1234, Work: 555-555-4321
   555-555-1234
   555-555-4321
```

7.8.4 Simple Data Wrangling Steps Used to Prepare Text for Training NLP and Generative AI Models

Data does not always come ready for analysis or training artificial intelligence (AI) models. It might be incorrect, in the wrong format or even missing. Industry experience has shown that data scientists spend as much as 80% of their time preparing data before analyzing it.[8] Preparing data for analysis is called data munging or data wrangling. From this point forward, we'll say data wrangling. Figure 7.22 uses `String` and regex capabilities to perform several common data-wrangling tasks that might be used to prepare text for training the large language models (LLMs) that generative AIs use to understand and respond to your prompts. Some of these include the tasks we listed in Section 7.7's Intro to Natural Language Processing (NLP)—at the Root of Generative AI. Others include:

- converting text to all lowercase letters (shown in Section 7.3.7),
- removing punctuation,
- collapsing consecutive whitespace characters to a single space,
- removing personally identifiable information (PII),
- normalizing numbers (sometimes specific numeric values are not relevant for analysis or training, so you might replace them with tags like `<NUMBER>` or `<NUM>`),
- removing numbers and
- removing special symbols.

8. Pragmatic Institute. "Overcoming the 80/20 Rule in Data Science." Accessed September 6, 2024. https://www.pragmaticinstitute.com/resources/articles/data/overcoming-the-80-20-rule-in-data-science/.

Whether these are performed on a particular corpus (body of text) typically depends on

- what an AI model needs to learn from the corpus or
- how the model needs to analyze it.

For example, you might eliminate stop words for a model that summarizes word frequencies for a corpus's most important words.

Keeping stop words could be important for a model that's learning sentence structure so it can respond to your prompts in complete, well-structured sentences, like those produced in generative AI applications.

Figure 7.22 performs several data wrangling steps on the text block `corpus` defined in lines 7–14, which intentionally contains extra whitepace. We've broken this example into pieces for discussion purposes.

Click here to view code image

```
1  // Fig. 7.22: Wrangling.java
2  // Simple data wrangling tasks to prepare text to train language models.
3  import java.util.Arrays;
4
5  public class Wrangling {
6     public static void main(String[] args) {
7        String corpus = """
8           In C++, an int  on one  machine  might be 16 bits     (2 bytes)
9           of   memory,    on another    32 bits (4 bytes), and on another
10          64 bits (8 bytes).    In Java,   int values are  always 32 bits
11          (4 bytes). The eight  primitive types in Java -- boolean, byte,
12          char, double, float, int, long and short -- are portable across
13          all     computer     platforms      that     support      Java.
14          """;
15       System.out.printf("INITIAL CORPUS:%n%s%n", corpus);
16
```

```
INITIAL CORPUS:
In C++, an int  on one  machine  might be 16 bits     (2 bytes)
of   memory,    on another    32 bits (4 bytes), and on another
64 bits (8 bytes).    In Java,   int values are  always 32 bits
(4 bytes). The eight  primitive types in Java -- boolean, byte,
char, double, float, int, long and short -- are portable across
all     computer     platforms      that      support      Java.
```

Fig. 7.22 | Simple data wrangling tasks to prepare text to train language models.

Lowercasing/Case Folding

A common data wrangling step is to convert an entire corpus to lowercase letters. Recall that Java compares `String`s lexicographically by default, so the words `"happy"` and `"Happy"` are different. However, they are still the same word and for a task like word-frequency counting, so we'd want to treat them as the same

word. You could use regular expressions to replace all uppercase letters with lowercase letters but, as you saw in Section 7.3.7, Java provides `String` method `toLowerCase` for that purpose. In this step's output, note that `toLowercase` changed "C++" to "c++."

Click here to view code image

```
17          // convert corpus to lowercase
18          String lowercased = corpus.toLowerCase();
19          System.out.printf("LOWERCASED CORPUS:%n%s%n", lowercased);
20
```

```
LOWERCASED CORPUS:
in c++, an int  on one  machine  might be 16 bits     (2 bytes)
of   memory,    on another    32 bits (4 bytes), and on another
64 bits (8 bytes).    in java,   int values are  always 32 bits
(4 bytes). the eight  primitive types in java -- boolean, byte,
char, double, float, int, long and short -- are portable across
all     computer     platforms      that     support      java.
```

Removing Punctuation

For some kinds of text analysis and training, you must remove punctuation to make the text easier to tokenize. Line 22 introduces a special character class that matches any Unicode punctuation character—**`\p{Punct}`**—and uses it in a `replaceAll` call to replace every punctuation character in the `String lowercased` with an empty `String` (`""`). Once again, note that we must precede a backslash in a regular expression with an extra backslash:

Click here to view code image

```
21          // remove punctuation
22          String noPunctuation = lowercased.replaceAll("\\p{Punct}", "");
23          System.out.printf("PUNCTUATION REMOVED:%n%s%n", noPunctuation);
24
```

```
PUNCTUATION REMOVED:
in c an int  on one  machine  might be 16 bits     2 bytes
of   memory    on another    32 bits 4 bytes and on another
64 bits 8 bytes    in java   int values are  always 32 bits
4 bytes the eight  primitive types in java  boolean byte
char double float int long and short  are portable across
all     computer     platforms      that      support      java
```

In this step's output, note that removing punctuation transformed "c

++" to "c." So in this series of data wrangling steps, "C++" became "c++" then "c." You should always perform wrangling operations cautiously.

Collapsing and Normalizing Whitespace

This example's original `String` was inconsistently spaced. It's common to normalize spacing by collapsing consecutive whitespace characters to a single space. In line 26, the regular expression `\s+` replaces any one or more whitespace characters with a single space, so any individual newline (`\n`) or tab (`\t`) character is also replaced with a space. After this operation, all tokens in the `String` are uniformly separated from one another with one space. Though we did not do so here, we also could call `strip` on the resulting `String` to remove any leading or trailing whitespace (there is none in this example).

Click here to view code image

```
25          // collapsing consecutive whitespace characters to one space
26          String spacingNormalized = noPunctuation.replaceAll("\\s+", " ");
27          System.out.printf("SPACING NORMALIZED:%n%s%n%n", spacingNormalized);
28
```

```
SPACING NORMALIZED:
in c an int on one machine might be 16 bits 2 bytes of memory on another 32
bits 4 bytes and on another 64 bits 8 bytes in java int values are always 32
bits 4 bytes the eight primitive types in java boolean byte char double float
int long and short are portable across all computer platforms that support java
```

Tokenizing

Now that we've performed several data wrangling steps on our initial text block, line 30 calls `String` method `split` to tokenize the text, and line 31 displays the resulting array of tokens, using `Arrays` method `toString` to create a comma-separated `String` representation of the tokens. Language models might use these tokens to perform a frequency analysis.

Click here to view code image

```
29          // tokenizing the text
30          String[] tokenized = spacingNormalized.split(" ");
31          System.out.printf("TOKENS:%n%s%n", Arrays.toString(tokenized));
32      }
33  }
```

```
TOKENS:
[in, c, an, int, on, one, machine, might, be, 16, bits, 2, bytes, of, memo-
ry, on, another, 32, bits, 4, bytes, and, on, another, 64, bits, 8, bytes,
in, java, int, values, are, always, 32, bits, 4, bytes, the, eight, primi-
tive, types, in, java, boolean, byte, char, double, float, int, long, and,
short, are, portable, across, all, computer, platforms, that, support, java]
```

Generative AI

1 Prompt genAIs with the code from Fig. 7.22. Ask them to modify the code to normalize numbers, by replacing them with tags `<NUMBER>`, then confirm the results by running the updated program.
2 Prompt genAIs for brief explanations and examples of other characters classes like `\p{Punct}` that you can use in Java.
3 Prompt genAIs for a brief Java tutorial on removing personally identifiable information (PII) form a corpus.

7.9 Objects-Natural Security Case Study: pMa5tfEKwk59dTvC04Ft1IFQz9mEXnkfYXZwx

You probably noticed that

- this chapter's last Objectives bullet,
- Section 7.9's title in the chapter outline and here, and
- the last subheading in Section 7.1

all look like gibberish. These are not mistakes—they're encrypted text. Here, we continue our objects-natural approach with a case study on encrypting and decrypting messages using objects of Java API classes provided by the

- **Java Cryptography Architecture (`java.security` package[9])** and

  9. "Package `java.security`," Accessed September 21, 2024. https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/security/package-summary.html.

- **Java Cryptography Extensions (`javax.crypto`[10] and**

related packages).

10. "Package `javax.crypto`," Accessed September 21, 2024. https://docs.oracle.com/en/java/javase/23/docs/api/java.base/javax/crypto/package-summary.html.

The Java Cryptography Architecture (JCA) defines the framework for using cryptography capabilities. The Java Cryptography Extensions enhance the JCA with advanced cryptography algorithms and other capabilities that industrial-strength applications require to securely transmit and store data.

Cryptography

Cryptography has been used for thousands of years[11,12] and is critically important in today's connected world. Every day, it's used behind the scenes to ensure that your Internet-based communications are private and secure. For example, most websites now use the HTTPS protocol to encrypt and decrypt your web interactions, as we do at `deitel.com`.

11. "Cryptography." Wikipedia. Wikimedia Foundation. Accessed September 6, 2024. https://en.wikipedia.org/wiki/Cryptography#History_of_cryptography_and_cryptanalysis.
12. Binance Academy. "History of Cryptography." Updated December 23, 2022. Accessed September 6, 2024. https://www.binance.vision/security/history-of-cryptography.

Ciphers

Julius Caesar used a simple **substitution cipher** to encrypt military communications.[13] His technique—known as the **Caesar cipher**[14]—replaces every letter in a communication (the **plaintext**) with the letter three ahead in the alphabet. So, A is replaced with D, B with E, C with F, …, X with A, Y with B and Z with C. Thus, the plaintext

13. "Caesar Cipher." Wikipedia. Wikimedia Foundation. Accessed September 6, 2024. https://en.wikipedia.org/wiki/Caesar_cipher.
14. For a fun way to play with the Caesar cipher and many other cipher techniques, check out the website https://cryptii.com/pipes/caesar-cipher, which is an online implementation of the open-source cryptii project: https://github.com/cryptii/cryptii.

```
Caesar Cipher
```

would be encrypted as

```
Fdhvdu Flskhu
```

The encrypted text is known as the **ciphertext**.

Symmetric-Key Cryptography

In this objects-natural case study, you'll work with **symmetric-key cryptography** (also called **private-key cryptography** or **secret-key cryptography**) in which the message sender's plaintext will be encrypted with a secret key to form ciphertext. The term "symmetric" comes from the fact that the receiver must use the same secret key to decode the ciphertext and restore the original plaintext. A problem with secret-key cryptography is that the security of the ciphertext is only as good as the security of the secret key, and several copies of that key are "floating around." In a Chapter 11 objects-natural case study, we'll discuss **asymmetric-key cryptography** in which different keys are used for encryption and decryption—this technique is often used to transmit the secret keys used in symmetric encryption. Asymmetric-key cryptography is much slower than symmetric key, so symmetric-key cryptography is preferred for encrypting and decrypting lengthy messages.[15]

15. "Public-key cryptography," Wikipedia. Wikimedia Foundation. Accessed September 6, 2024. https://en.wikipedia.org/wiki/Public-key_cryptography.

Brief Intro to Advanced Encryption Standard (AES)

In this objects-natural case study, you'll work with a simplified version of the **Advanced Encryption Standard (AES)**—one of the most widely used symmetric-key cryptography algorithms and the choice of the U.S. government for securing communications.[16] AES supports secret keys of 128, 192 or 256 bits—the more bits, the more secure the key and the more computing power the algorithm requires to perform encryption and decryption. Depending on an application's security requirements, you might choose keys with fewer bits for better performance.

16. "Advanced Encryption Standard," Wikipedia. Wikimedia Foundation. Accessed September 6, 2024. https://en.wikipedia.org/wiki/Advanced_Encryption_Standard.

For Demo Purposes Only: Reproducible Secret Keys

Sometimes, you may want to guarantee **reproducibility** of a random sequence. Later in this section, we'll show how to do this with the `SecureRandom` class's `setSeed` method. Reproducibility is also useful for debugging applications with random numbers.

For our encryption example, we'll use reproducible secret keys so you can decrypt the encrypted text you saw earlier in this chapter and in this section's title. **Industrial-strength applications should use randomly generated secret keys that are not reproducible.**

To make our secret keys reproducible so you can decrypt our sample ciphertext, we'll generate secret keys using a `SecureRandom` random-number generator that's seeded with the byte representation of the following arbitrary 11-character `String`—**you will not seed the `SecureRandom` random-number generator in industrial-strength applications**:

```
XMWUJBVYHXZ
```

The secret key itself will have 256 bits. To decrypt the ciphertext, you must know the secret key used to create the ciphertext.

Encrypting and Decrypting Text with AES

Class `CryptographyDemo` (Fig. 7.23) uses AES to encrypt and decrypt messages. Lines 3–8 `import` the features we use from packages `javax.crypto`, `java.security` and `java.util`.

Click here to view code image

```
1  // Fig. 7.23: CryptographyDemo.java
2  // Encrypting and decrypting messages with AES.
3  import javax.crypto.Cipher;
4  import javax.crypto.KeyGenerator;
5  import javax.crypto.SecretKey;
6  import java.security.SecureRandom;
7  import java.util.Base64;
8  import java.util.Scanner;
9
10 public class CryptographyDemo {
```

Fig. 7.23 | Encrypting and decrypting messages with AES.

Method `main`

The `main` method handles the user interactions. Lines 14–15 prompt for and input the `plaintext` to encrypt:

Click here to view code image

```
11    public static void main(String[] args) {
12       var scanner = new Scanner(System.in);
```

```
13
14          System.out.println("Enter the text to encrypt:");
15          String plaintext = scanner.nextLine();
16
```

```
Enter the text to encrypt:
Welcome to encryption and decryption using AES in Java!
```

Lines 17–19 prompt for and input the `String` used to help create a reproducible key so you can decrypt the ciphertext in Section 7.9's title—again, this is for demo purposes only:

Click here to view code image

```
17          System.out.println(
18             "\nEnter the secret key seed (for reproducibility):");
19          String seedString = scanner.nextLine();
20
```

```
Enter the secret key seed (for reproducibility):
XMWUJBVYHXZ
```

Lines 22–39 in `main` are enclosed in a `try` block because many cryptography API methods throw exceptions that you're required to "`catch` or declare"—we'll discuss this requirement in Section 10.4. Lines 41–46 `catch` this program's potential exceptions and display an error message. Line 23 calls our `generateKey` method (lines 52–61) to obtain the secret key we'll use to encrypt the `plaintext` `String` and, later, decrypt the ciphertext `String`—you'll soon see that `generateKey` uses a Java API class to create the secret key:

Click here to view code image

```
21          try { // encryption/decryption might throw exceptions
22             // generate a reproducible AES key from the seedString
23             SecretKey secretKey = generateKey(seedString);
24             System.out.printf("%nSecretKey:%n%s%n",
25                Base64.getEncoder().encodeToString(secretKey.getEncoded()));
26
```

```
SecretKey:
8ew8jBJVSpkRtG4zorwxd34JBpUxEdOt+ec0QTfr7cE=
```

Secret keys and ciphertext are stored as `byte` arrays. When these are transmitted over networks (like the Internet) and between

systems that handle the raw bytes differently, the secret keys and ciphertext can get corrupted. However, those systems typically understand lowercase letters, uppercase letters, digits and various special symbols. So, raw bytes are often transformed into character representations for transmission. One format for this is called **Base64**—named for the **Base64 alphabet's** 64 characters, consisting of 52 letters (`A`–`Z` and `a`–`z`), 10 digits (`0`–`9`) and two special symbols (`+` and `/`), all of which are printable and easily understood by the world's computer systems.[17] Lines 24–25 convert the secret key's byte representation to Base64 format as follows:

- **`SecretKey`'s `getEncoded` method** returns a `byte` array representation of the key.
- **`Base64`'s `getEncoder` method** returns a `Base64` object that can convert a `byte` array to a `String` representation containing only Base64-alphabet characters. We then use the `Base64` object's **`encodeToString` method** to convert the `byte` array to a `String`. This enables us to display the secret key (and momentarily, the ciphertext) for demo purposes.[18]

17. "Base64," Wikipedia. Wikimedia Foundation. Accessed September 6, 2024. https://en.wikipedia.org/wiki/Base64.
18. "Class Base64." Accessed September 21, 2024. https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/util/Base64.html.

Line 28 calls our `encrypt` method (lines 64–75) to encrypt the `plaintext` using the `secretKey`, and line 29 displays the resulting `ciphertext`, which our method returns as a `Base64` encoded `String`:

Click here to view code image

```
27          // encrypt the plaintext
28          String ciphertext = encrypt(plaintext, secretKey);
29          System.out.printf("%nEncrypted:%n%s%n", ciphertext);
30
```

```
Encrypted:
Cz9o3YoFcX+yebfAOuMPMi+UZz2bc5jJdstfzapaNKwJldZuQH1f3QKNKrh9Zh4aB5tRbns6ZL9b
XF8Bq2YNeQ==
```

Lines 32–33 call method `decrypt` (lines 78–89) to decrypt the `ciphertext` using the `secretKey` then display the resulting plaintext, which matches the original message:

Click here to view code image

```
            // decrypt the text
            System.out.printf("%nDecrypted:%n%s%n",
               decrypt(ciphertext, secretKey));
```

```
Decrypted:
Welcome to encryption and decryption using AES in Java!
```

Lines 36–37 prompt for and input existing ciphertext, so you can decrypt this section's title. For your convenience, we placed that text in the file `encrypted.txt` in this example's folder so you can copy and paste it into the program. Lines 38–39 call our `decrypt` method to decrypt that `ciphertext` using the `secretKey` then display the resulting plaintext for the encrypted part of Section 7.9's title.

Click here to view code image

```
            // decrypt ciphertext entered by the user
            System.out.println("\nEnter the ciphertext to decipher:");
            ciphertext = scanner.nextLine();
            System.out.printf("%nDecrypted:%n%s%n",
               decrypt(ciphertext, secretKey));
         }
         catch (Exception ex) {
            System.out.printf("""
               An exception occurred during encryption/decryption:
               %s%n
               """, ex);
         }
      }
```

```
Enter the ciphertext to decipher:
pMa5tfEKwk59dTvC04Ft1IFQz9mEXnkfYXZwxk4ujGE=

Decrypted:
Symmetric-Key Cryptography
```

Method `generateKey`

The `generateKey` method (lines 52–61) generates a 256-bit AES `SecretKey` using its argument to seed a `SecureRandom` generator for

reproducibility. This allows our example to generate the same `SecretKey` from the same `seed`. This is useful for testing and demonstration purposes and will enable you to decrypt the ciphertext in this section's title. **`SecretKey`s for industrial-strength encryption should be randomly generated so they are not reproducible.** The clause "`throws Exception`" after `main`'s parameter list and before its body is required for this example to compile. Many encryption and decryption methods can cause exceptions the Java compiler requires us to handle. We'll explain the "`throws Exception`" clause in detail in Section 10.3.

Click here to view code image

```
49      // Generate and return a SecretKey. Note: We are seeding a
50      // SecureRandom object for reproducibility so you can decipher this
51      // chapter's encrypted text. Don't do this in production code.
52      public static SecretKey generateKey(String seed) throws Exception {
53         // get a secure random number generator
54         var random = SecureRandom.getInstance("SHA1PRNG");
55         random.setSeed(seed.getBytes()); // for demo reproducibility
56
57         // get an AES SecretKey generator
58         var keyGen = KeyGenerator.getInstance("AES");
59         keyGen.init(256, random); // set up for 256-bit keys
60         return keyGen.generateKey(); // generate and return new SecretKey
61      }
62
```

The `generateKey` method operates as follows:

- Line 54 gets a `SecureRandom` object configured to use the `"SHA1PRNG"` random-number generation algorithm, which can be seeded for reproducibility.
- Line 55 seeds the generator with `seed String`'s `byte` array representation (returned by **`String` method `getBytes`**). Seeding ensures the same random values are generated each time we use the same `seed String`, making the `SecretKey` generation reproducible, which we do only for demo purposes.
- Line 58 uses the **`KeyGenerator` `static` method `getInstance`** to get a `KeyGenerator` configured for generating AES secret keys.
- Line 59 calls **`KeyGenerator` method `init`** with the AES **key size** and the seeded `SecureRandom` instance `random`—256 bits

produces the most secure AES keys. For better security, use the one-argument `init` method that receives only the key size, or provide as `init`'s second argument a fully randomized (i.e., not seeded) `SecureRandom` object so the `KeyGenerator` can generate truly random keys.

- Finally, line 60 calls the **`KeyGenerator` method `generateKey`** to get and return a `SecretKey` for encrypting and decrypting messages.

Method `encrypt`

The `encrypt` method (lines 64–75) receives the `plaintext` to encrypt and a `SecretKey`, performs AES encryption and returns a `Base64` encoded version of the ciphertext. Again, the clause "`throws Exception`" is required for the code to compile.

Click here to view code image

```
63      // encrypt a String using AES
64      public static String encrypt(
65         String plaintext, SecretKey secretKey) throws Exception {
66
67         var cipher = Cipher.getInstance("AES"); // get a Cipher object
68         cipher.init(Cipher.ENCRYPT_MODE, secretKey); // configure to encrypt
69
70         // encrypt the byte[] representation of plaintext
71         byte[] encrypted = cipher.doFinal(plaintext.getBytes());
72
73         // Base64 encode encrypted as a ciphertext String and return
74         return Base64.getEncoder().encodeToString(encrypted);
75      }
76
```

Method `encrypt` operates as follows:

- Line 67 calls the **`Cipher`[19] class `static getInstance` method** to get a `Cipher` object we can use to encrypt plaintext or decrypt ciphertext. This `Cipher` object will use the AES algorithm.

  19. "Class `Cipher`." Accessed September 21, 2024. https://docs.oracle.com/en/java/javase/23/docs/api/java.base/javax/crypto/Cipher.html.

- Line 68 initializes the `Cipher` object for encryption (**`Cipher.ENCRYPT_MODE`**), using the `secretKey`.
- Line 71 gets the `plaintext`'s `byte` array representation and

passes it to the `Cipher` object's `doFinal` method, which encrypts the data and returns it as a `byte` array.

- Line 74 Base64 encodes `encrypted` as a `String` and returns the ciphertext.

Method `decrypt`

The `decrypt` method (lines 78–89) receives the Base64-encoded `ciphertext` to decrypt and the `SecretKey` used to encrypt the data, then performs AES decryption and returns the plaintext `String`. Again, the clause "`throws Exception`" is required for the code to compile.

Click here to view code image

```
77      // decrypt an AES-encrypted String
78      public static String decrypt(
79         String ciphertext, SecretKey secretKey) throws Exception {
80
81         var cipher = Cipher.getInstance("AES"); // get a Cipher object
82         cipher.init(Cipher.DECRYPT_MODE, secretKey); // configure to decrypt
83
84         // Base64 decode ciphertext to byte array
85         byte[] ciphertextBytes = Base64.getDecoder().decode(ciphertext);
86
87         // create plaintext String from ciphertextBytes and return it
88         return new String(cipher.doFinal(ciphertextBytes));
89      }
90   }
```

Method `decrypt` operates as follows:

- As in the `encrypt` method, line 81 calls the `Cipher` class's `static getInstance` method to get a `Cipher` that we'll use for decryption with `"AES"`.
- Line 82 initializes the `Cipher` object for decryption (`Cipher.DECRYPT_MODE`) using the `secretKey`.
- Line 85 decodes the Base64-encoded `ciphertext String` into the `byte` array `ciphertextBytes`. First, we call `Base64`'s `getDecoder` method to get an object that can convert a Base64-encoded `String` to a `byte` array. Then, we call that object's `decode` method to perform the conversion.
- To decrypt the `bytes` in `ciphertextBytes`, line 88 calls `Cipher`'s `doFinal` method, passing `ciphertextBytes`. The

`doFinal` method returns an array of decrypted `byte`s.

- Line 88 creates a new `String` from the `byte` array and returns the original plaintext.

Weakness in Secret-Key Cryptography—and a Look to Public-Key Cryptography

Secret-key cryptography has a weakness—the ciphertext is only as secure as the secret key. Ciphertext can be decrypted by anyone who discovers or steals the secret key. In a Chapter 11 objects-natural case study, we'll complete our introduction to Java cryptography with a discussion of **public-key cryptography**. This technique performs encryption with a public key known to every sender who may want to send a secret message to a particular receiver. The public key can be used to encrypt messages but not decrypt them. The messages can be decrypted only with a paired private key known only to the receiver. We'll use public-key cryptography to securely transmit the secret keys used in secret-key cryptography.

A Note about Cryptography and Computing Power

Ideally, ciphertext should be impossible to "break"—that is, it should be impossible to determine the plaintext from the ciphertext without the decryption key. For various reasons, that goal is not realizable. So, designers of cryptography schemes settle for making them extraordinarily difficult to break. Unfortunately, today's increasingly powerful computers (with quantum computers on the horizon) are making it possible to break many of the encryption schemes the world has used confidently over the last few decades.[20,21] Cryptography is at the root of the blockchain technologies that implement cryptocurrencies such as Bitcoin.[22] The phenomenally powerful computers that quantum computing will enable are putting cryptography schemes and cryptocurrencies at risk.[23,24] The cryptocurrency community is working on these challenges.[25,26,27]

20. Martin Giles, "Inside the Quest for Unbreakable Encryption," MIT Technology Review, October 19, 2023, Accessed September 6, 2024. https://www.technologyreview.com/2023/10/19/1081389/unbreakable-encryption-quantum-computers-cryptography-math-problems/.

21. Jason Dorrier, "Quantum Computers Could Crack Encryption Sooner Than Expected With New Algorithm," Singularity Hub, October 2, 2023, Accessed September 6, 2024. https://

singularityhub.com/2023/10/02/quantum-computers-could-crack-encryption-sooner-than-expected-with-new-algorithm/.

22. "Cryptocurrency." Accessed September 6, 2024. https://www.investopedia.com/terms/c/cryptocurrency.asp.

23. "The Impact of Quantum Computing on Present Cryptography." Accessed September 6, 2024. https://arxiv.org/pdf/1804.00200.pdf.

24. "Quantum Computing and its Impact on Cryptography." Accessed September 6, 2024. https://www.cryptomathic.com/news-events/blog/quantum-computing-and-its-impact-on-cryptography.

25. "How Should Crypto Prepare for Google's 'Quantum Supremacy'?" Accessed September 6, 2024. https://www.coindesk.com/how-should-crypto-prepare-for-googles-quantum-supremacy.

26. "Here's Why Quantum Computing Will Not Break Cryptocurrencies." Accessed September 6, 2024. https://www.forbes.com/sites/rogerhuang/2020/12/21/heres-why-quantum-computing-will-not-break-cryptocurrencies/.

27. "How the Crypto World Is Preparing for Quantum Computing, Explained." Accessed September 6, 2024. https://cointelegraph.com/explained/how-the-crypto-world-is-preparing-for-quantum-computing-explained.

Generative AI

1 Prompt genAIs to explain why developers don't automatically use 256-bit keys.

2 Prompt genAIs to assess the risks posed to today's cryptography schemes by emerging quantum computers. Ask them to explain what is being done in the cryptography community today to meet those challenges.

7.10 Wrap-Up

This chapter presented Java's `String`, `StringBuilder` and `Character` classes. You saw various ways to create `String` objects, then used `String` methods to determine a `String`'s length, get characters and character arrays from `String`s, compare `String`s, locate characters and substrings in `String`s, extract substrings, concatenate `String`s, replace characters in `String`s and change the case of a `String`. You saw that `String`s are immutable, so each `String` operation that appears to modify a `String`'s contents copies the contents, performs the modification on the copy and returns a new `String` containing the result.

We used modifiable objects of class `StringBuilder` to dynamically create and manipulate `String` content without creating unnecessary `String` objects. We used class `Character` to test characters for various attributes. We converted between characters and their corresponding Unicode integer values and vice versa. We used the `String` method `split` to tokenize `String`s—breaking text into sentences and words is a key aspect of helping computers understand natural language.

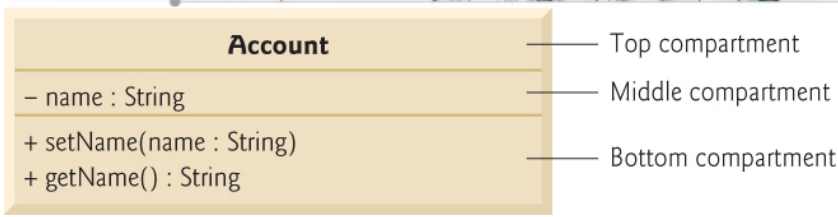
We introduced natural language processing (NLP), which helps computers understand, analyze and process text, and serves as a foundation for generative AI. We explained how we used Grammarly and generative AI's NLP features to tune our writing, evaluate our chapters breadth and depth of coverage, verify the content and suggest improvements.

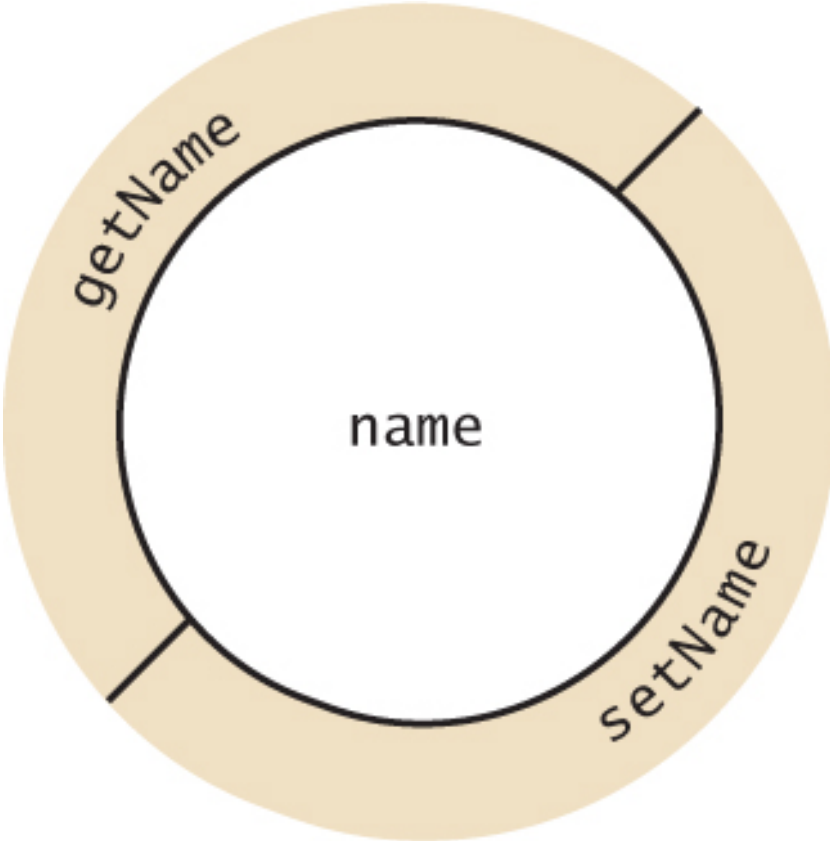
We continued our objects-natural approach with two case studies. The first introduced regular expressions—crucial in today's data-rich applications for cleaning and preparing data for analysis, data mining and transforming data to other formats. You used `String` methods and the `java.util.regex` package's `Matcher` and `Pattern` classes to perform these tasks. These capabilities are frequently used to validate data. They are also used in natural-language processing, preparing data for training generative AIs and helping generative AI models create responses to your prompts. In Chapter 11, we'll do more with NLP.

In the second objects-natural case study, we used the Java Cryptography Architecture (JCA) and Java Cryptography Extensions to introduce symmetric cryptography using the popular and powerful AES encryption algorithm. In that case study, you saw how to generate secret keys, prepare plaintext for encryption, configure a `Cipher` for encryption, encrypt plaintext to create ciphertext, configure a `Cipher` for decryption and decrypt ciphertext to restore the original plaintext. Java provides extensive security and cryptography features. To study these in depth, refer to Oracle's Security Developer's Guide.[28] We'll continue our Objects-Natural Security Case Study with an introduction to asymmetric encryption in Chapter 11, Files, I/O Streams, JSON Serialization & CSV Files.

28. Oracle, "Security Developer's Guide," Accessed September 21, 2024, https://docs.oracle.com/en/java/javase/23/security/index.html.

In the next chapter, you'll create custom classes. You'll see that each new class you create is a type you can use to declare variables and create objects.

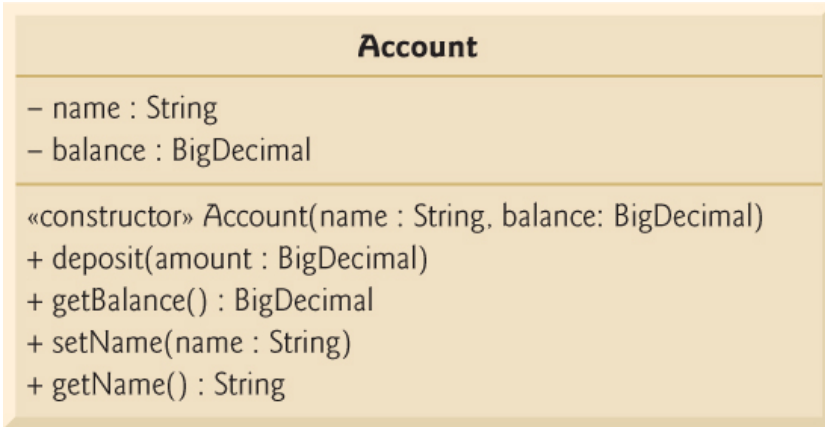

Account
– name : String
+ setName(name : String)
+ getName() : String
Top compartment
Middle compartment
Bottom compartment



Account

– name : String

«constructor» Account(name: String)
+ setName(name: String)
+ getName() : String



Real-World Modeling with Custom Classes

Objectives

In this chapter, you'll:

- Declare classes and create objects of these classes.
- Call methods to make objects perform their operations.
- Compare a method's local variables with a class's instance variables.
- Initialize objects with constructors.
- Learn why classes are a natural way to model real-world

things and abstract entities.

- `throw` exceptions to indicate a runtime problem occurred.
- Delegate work from one constructor to another in the same class via `this`.
- Define `static` class members.
- Import `static` class members.
- Create sets of named constants with `enum`s.
- Declare `enum` constants with parameters.
- Use `record`s to create immutable objects primarily for storing related data items.
- Use `record` objects with pattern-matching `switch` expressions.
- Continue using generative AI to leverage your Java learning experience.

Outline

8.1 Introduction

8.2 Instance Variables, *set* Methods and *get* Methods

8.2.1 `Account` Class with an Instance Variable, and *set* and *get* Methods

8.2.2 `AccountTest` Class That Creates and Uses an `Account` Object

8.2.3 Compiling and Executing a Program with Multiple Classes

8.2.4 `Account` UML Class Diagram

8.2.5 Notes on Class `AccountTest`

8.2.6 Software Engineering with `private` Instance Variables and `public` *set* and *get* Methods

8.3 Default and Explicit Instance Variable Initialization

8.4 `Account` Class: Initializing Objects with Constructors

8.4.1 Declaring an `Account` Constructor for Custom Object Initialization

8.4.2 Class `AccountTest`: Initializing New `Account` Objects

8.5 `Account` Class with a Balance

8.5.1 `Account` Class with a `BigDecimal balance` Instance Variable

8.5.2 `AccountTest` Class
8.6 Case Study: Card Shuffling and Dealing Simulation
8.7 Case Study: `Time` Class
8.8 Controlling Access to Members
8.9 Referring to the Current Object's Members with the `this` Reference
8.10 Case Study: `Time` Class Overloaded Constructors
8.11 Default and No-Argument Constructors
8.12 Notes on *Set* and *Get* Methods
8.13 Composition
8.14 `enum` Types
8.15 Garbage Collection
8.16 `static` Class Members
8.17 `static` Import
8.18 `final` Instance Variables
8.19 Package Access
8.20 `record` Classes
8.20.1 Pattern Matching for `switch` Expressions
8.20.2 Decomposing `record`s in `switch` Expressions
8.21 Wrap-Up

8.1 Introduction[1]

1. This chapter depends on the terminology and concepts of object-oriented programming introduced in Section 1.5, Introduction to Object Technology.

In the preceding chapters, you worked with existing classes, objects and methods. For example, you used the predefined standard output object `System.out`, invoking `print`, `println` and `printf` to display information on the screen. You used `Scanner` objects to input data from the user. In the Objects-Natural Case Studies, you:

- manipulated objects of class `String`,
- performed calculations with super-sized integers (of class `BigInteger`),

- performed precise monetary calculations (with class `BigDecimal`),
- manipulated dates and times (with the Java Date/Time API),
- processed dynamically resizable collections of objects (with class `ArrayList`),
- matched patterns in text with regular expressions (using classes `String`, `Pattern` and `Matcher`), and
- used secret-key cryptography to encrypt and decrypt data (with classes `Cipher`, `KeyGenerator` and `SecretKey`). In Chapter 11, Files, I/O Streams, JSON Serialization & CSV Files, Objects-Natural Case Study, you'll explore public-key cryptography.

Throughout the book, you'll use many more preexisting classes and objects. This is one of the great strengths—and challenges—of learning object-oriented programming with Java.

In this chapter, you'll create custom classes. Each new class you create is a type you can use to declare variables and create objects. You can declare new classes as needed; this is one reason why Java is known as an extensible language.

Sections 8.2–8.5 present a case study on creating and using a simple bank-account class. When we complete the case study, the class will maintain `name` and `balance` attributes (as instance variables) and provide methods for querying the balance (`getBalance`) and making deposits that increase the balance (`deposit`). As an exercise, you can add a `withdraw` method that decreases the balance.

Next, we present a card-shuffling-and-dealing simulation. Then, using a `Time` class case study and several additional classes, we continue with a deeper look at building classes, controlling access to class members and creating constructors to initialize objects. You'll validate data and `throw` an exception to indicate that a problem has occurred—Section 6.12 discussed `catch`ing exceptions. We use the `this` keyword to enable one constructor to delegate work to another of the same class. We examine the use of *set* and *get* methods for changing and retrieving an object's instance variable values. We discuss composition—a capability that allows a class to have references to objects of other classes as members.

Section 5.5 introduced the basic `enum` type to declare a group of named constants. Here, we discuss the relationship between `enum` types and classes, demonstrating that an `enum` type can have

constructors, methods and fields, but an `enum` type defines all objects of its type in advance. The chapter also discusses `static` class members and `final` instance variables in detail. We discuss how Java cleans up unused objects, and we show a special relationship among classes in the same package.

We introduce `record` classes for creating immutable objects that conveniently store related data items. As you'll see, `record` classes autogenerate various common methods. We show how to use `record` objects with `switch` expressions to perform operations based on the controlling-expression object's `record` class type and how to extract a `record` object's data items into variables for use in that `case`.

In later chapters, we discuss two additional types of classes—nested classes and anonymous inner classes. Many programs in this book will consist of two or more classes. If you become part of a development team in industry, you might work on applications containing hundreds or even thousands of classes.

8.2 Instance Variables, *set* Methods and *get* Methods

Let's create custom classes `Account` (Fig. 8.1) and `AccountTest` (Fig. 8.2). `AccountTest`'s `main` method will create and use an `Account` object and demonstrate its capabilities.

8.2.1 `Account` Class with an Instance Variable, and *set* and *get* Methods

Different accounts typically have different account-holder names. For this reason, our first version of the `Account` class (Fig. 8.1) contains a `name` instance variable. A class's instance variables maintain data for each object (that is, instance) of the class. Later, we'll add the instance variable `balance` to store how much money is in the account. Class `Account` provides two methods—`setName` stores a name in an `Account` object, and `getName` obtains a name from an `Account` object.

Click here to view code image

```
1  // Fig. 8.1: Account.java
2  // Account class with a name instance variable
```

```
3  // and methods to set and get its value.
4
5  public class Account {
6     private String name; // instance variable
7
8     // method to set the name
9     public void setName(String name) {
10       this.name = name; // store the name
11    }
12
13    // method to retrieve the name
14    public String getName() {
15       return name; // return value of name to caller
16    }
17 }
```

Fig. 8.1 | `Account` class with a `name` instance variable and methods to *set* and *get* its value.

Class Declaration

The class declaration begins in line 5:

```
public class Account {
```

The keyword `public` is an **access modifier**. We'll declare most classes `public`—a first step in making them reusable.[2] Each `public` class declaration must reside in a file named as the class name followed by the `.java` filename extension; otherwise, a compilation error will occur. Thus, `public` classes `Account` and `AccountTest` (Fig. 8.2) must be declared in the separate files `Account.java` and `AccountTest.java`, respectively.

2. In Chapter 21, Java Platform Module System, we'll discuss using packages and modules to create reusable classes that can be imported and used by other programs.

Every class declaration contains the keyword `class` followed immediately by the class's name—in this case, `Account`. Class name are identifiers. By convention, they use camel-case naming and begin with an initial uppercase letter. You enclose a class's body in braces, as in lines 5 and 17 of Fig. 8.1.

Instance Variable `name`

Recall from Section 1.4 that an object has attributes, implemented as instance variables and carried with it throughout its lifetime. Each object (instance) has a copy of its class's instance variables.

These are declared inside the class declaration but outside the class's method bodies. Line 6

Click here to view code image

```
private String name; // instance variable
```

declares the `String` instance variable `name` outside the `setName` (lines 9–11) and `getName` (lines 14–16) methods' bodies. Each `Account` object has its own `name`. Because `name` is an instance variable, each of the class's instance methods can manipulate it. We prefer to list a class's instance variables first in its body so we know their names and types before using them in the class's methods. Technically, you can place them anywhere in the class outside its method declarations, but scattering the instance variables can lead to hard-to-read code.

Access Modifiers `public` and `private`

You declare most instance variables with the **`private` access modifier** (line 6). Variables or methods declared `private` are accessible only to methods of the class that declares them. In this example, `Account`'s methods `setName` and `getName` can use `name`. You'll soon see that this presents powerful software engineering opportunities.

`setName` Method of Class `Account`

Previously, you declared only `static` methods in each class. A class's non-`static` methods are known as **instance methods**. Method `setName`'s declaration (lines 9–11) specifies the `String` parameter `name`, representing the name to store in an `Account` object's `name` instance variable. The return type `void` indicates that `setName` does not return a value when it completes execution. Recall that a method's parameters and any variables declared in its body are local variables usable only in the method's body.

Method `setName`'s body (line 10) stores a name in an `Account` object by assigning the `name` parameter's value (a `String`) to the class's `name` instance variable. If a method contains a local variable (the parameter in line 9) with the same name as an instance variable (line 6), the local variable shadows (Section 5.6) the instance variable in the method's body.[3] On the left side of the assignment in line 10, we use the **`this` keyword** to explicitly refer to the shadowed instance variable. In each instance method, Java creates a reference named `this`, which refers to the object that

called the method.

3. IDEs will generally warn you about shadowing variables.

We could have avoided the need for `this` by choosing a different parameter name in line 9 but using `this`, as shown in line 10, is a widely accepted industry practice to minimize the proliferation of identifier names.

`getName` Method of Class `Account`

The `getName` method (lines 14–16) returns an `Account` object's `name` to its caller. The method returns a `String` and has an empty parameter list, so it does not require additional information to perform its operation. Line 15 uses the `name` instance variable directly—it is not shadowed in this method so there is no need to qualify it with "`this.`" In Fig. 8.2, we'll create and manipulate an `Account` object.

Generative AI

1 Prompt genAIs to explain why a class's instance variables must be declared outside the bodies of its methods.

2 Prompt genAIs, asking where Java programmers typically declare instance variables in a class declaration.

8.2.2 `AccountTest` Class That Creates and Uses an `Account` Object

Let's create an `Account` object and call its methods. As you know, a Java program begins executing in the `main` method. Class `Account` cannot execute by itself because it does not contain a `main` method, so if you try to execute class `Account` with the command

```
java Account
```

you'll get the error:

Click here to view code image

```
Error: Main method not found in class Account, please define the
main method as:
   public static void main(String[] args)
```

To fix this, you we'll declare a separate class containing a `main`

method. Though, we could include a `main` method in class `Account`, this is not typical for classes used to create objects. An `Account` is not a program. Rather, it's what an account-management system might use to represent and manipulate `Account`s.

Driver Class `AccountTest`

A person can drive a car by telling it what to do (go faster, go slower, turn left, turn right, etc.)—without knowing how the car's internal mechanisms work. Similarly, a method (such as `main`) can "drive" an `Account` object by calling its methods—without knowing how the class's internal mechanisms work. In this sense, the class containing `main` is known as a **driver class**. To help you prepare for the larger programs you'll encounter later in this book and industry, we define the `AccountTest` class containing `main` in the file `AccountTest.java` (Fig. 8.2). Once `main` begins executing, it may call other methods in this and other classes, such as `Account`. Class `AccountTest`'s `main` method creates one `Account` object and calls its `getName` and `setName` methods.

Click here to view code image

```
1  // Fig. 8.2: AccountTest.java
2  // Creating and manipulating an Account object.
3  import java.util.Scanner;
4
5  public class AccountTest {
6     public static void main(String[] args) {
7        // create a Scanner to obtain input from the user
8        var input = new Scanner(System.in);
9
10       // create an Account object and assign it to myAccount
11       var myAccount = new Account();
12
13       // display initial value of name (null)
14       System.out.printf("Initial name is: %s%n%n", myAccount.getName());
15
16       // prompt for and read name
17       System.out.println("Please enter the name:");
18       String theName = input.nextLine(); // read a line of text
19       myAccount.setName(theName); // put theName in myAccount
20       System.out.println(); // outputs a blank line
21
22       // display the name stored in object myAccount
23       System.out.printf("Name in object myAccount is:%n%s%n",
24          myAccount.getName());
25    }
```

```
26 }
```

```
Initial name is: null

Please enter the name:
Sofia Lopez

Name in object myAccount is:
Sofia Lopez
```

Fig. 8.2 | Creating and manipulating an `Account` object.

Instantiating an Object—Keyword `new` and Constructors

Line 11 uses a class instance creation expression to create a new `Account` object and assigns it to `myAccount`, using local-variable type inference to determine `myAccount`'s type. Recall that when `new` creates an object of a class, it calls the class's constructor to initialize the instance variables. In Section 8.4, you'll define a custom constructor with an argument so you can specify an initial value for an `Account` object's `name` instance variable. For now, we simply leave the parentheses empty.

Calling Class `Account`'s `getName` Method

Line 14 displays the initial name, obtained by calling the `myAccount` object's `getName` method with empty parentheses because `getName` does not require arguments.

`null`—the Default Initial Value for `String` Variables

The first line of the output shows the name "`null`." Unlike local variables, which are not automatically initialized, every instance variable has a **default initial value** if you do not specify one. So, you do not need to initialize instance variables explicitly before using them—unless they require non-default values. For a `String` or other reference-type instance variable, the default is `null`.

Getting the Name from the User

Lines 17–18 prompt for and input a name, assigning it to the local variable `theName`. You enter the name and press *Enter* to submit it to the program. `Scanner`'s `nextLine` method reads characters (including whitespace characters, such as the blank in `"Sofia`

`Lopez"`) until it encounters the newline, then returns a `String` containing the characters up to, but not including, the newline, which is discarded.

Recall that `Scanner`'s `next` method reads the next word. When you press *Enter* after inputting some text, `next` reads characters until it encounters a whitespace character, then returns a `String` containing the characters up to, but not including, the whitespace character, which is discarded. Information after the first whitespace character is not lost—subsequent statements can read the remaining input by calling the `Scanner`'s methods later.

Calling Class `Account`'s `setName` Method then Displaying the New Name

Line 19 calls `myAccount`'s `setName` method with `theName` as the argument. Lines 23–24 call `getName` again to get then display the name entered by the user in line 18, showing that `setName` modified the `Account` object's `name` in line 19.

Generative AI

1 Prompt genAIs asking them to explain why an `Account` class meant to create and manipulate objects should not also contain a `main` method.

8.2.3 Compiling and Executing a Program with Multiple Classes

Let's compile `Account.java` and `AccountTest.java` so we can execute the app. First, change to the folder containing the app's source-code files. Then, execute the following command to compile both classes:

Click here to view code image

```
javac Account.java AccountTest.java
```

If the folder contains only this program's files, you can compile both files as follows:

```
javac *.java
```

The **asterisk (`*`) wildcard** tells the compiler to compile all files

ending with the ".java" filename extension in the current folder. If both classes compile without errors, you can run the program with the command

```
java AccountTest
```

Running the .java Files Directly

Recall from Section 1.7 that you can conveniently test Java programs without compiling them first. As of Java 22, if you enter the command

```
java AccountTest.java
```

the JVM will compile `AccountTest.java` in memory. While doing so, it will recognize that `AccountTest.java` depends on `Account.java` and also compile `Account.java` in memory—neither class's bytecodes will be stored in `.class` files. Then, the JVM will call `AccountTest`'s `main` method to execute the program.

8.2.4 `Account` UML Class Diagram

We'll use UML class diagrams to help you visualize a class's attributes and operations. In industry, UML diagrams help systems designers specify a system in a concise, graphical, programming-language-independent manner before programmers implement the system in a specific programming language. The following diagram presents a **UML class diagram** for class `Account` (Fig. 8.1).

Top Compartment

In the UML, we represent each class as a rectangle with three compartments. The top compartment contains the class name `Account`, centered horizontally in boldface type.

Middle Compartment

The middle compartment contains the class's attribute `name`, corresponding to the instance variable of the same name in Java. The instance variable `name` is `private` in Java, so the UML class diagram lists a minus sign (–) access modifier before the attribute name. Following the attribute name are a colon and the attribute type—in this case, `String`.

Bottom Compartment

The bottom compartment contains the **class's operations**, `setName` and `getName`, corresponding to the methods of the same names in Java. The UML models an operation by listing the operation name preceded by an access modifier, in this case `+ getName`. This plus sign (+) indicates that `getName` is a public operation in the UML (because it's a `public` method in Java). Operation `getName` does not have any parameters, so the parentheses following the operation name in the class diagram are empty, just as they are in the method's declaration in line 14 of Fig. 8.1. Operation `setName`, also a public operation, has a `String` parameter called `name`.

Return Types

The UML indicates the return type of an operation by placing a colon and the return type after the parentheses following the operation name. `Account` method `getName` (Fig. 8.1) has a `String` return type. Method `setName` does not return a value (because it returns `void` in Java), so the UML class diagram does not specify a return type after the parentheses.

Parameters

The UML models a parameter differently from Java. In the parentheses after the operation name, we list the parameter name, followed by a colon and the parameter type. The UML has its own data types similar to those of Java, but for simplicity, we use the Java data types. `Account` method `setName` (Fig. 8.1) has a `String` parameter named `name`, so the diagram lists `name : String` between the parentheses following the method name.

Generative AI

1 Prompt genAIs with Fig. 8.1's code asking them to produce the corresponding UML class diagram.
2 Prompt genAIs with class `Account`'s UML class diagram, asking it to generate the corresponding Java class declaration. You can find the class diagram's image file with this chapter's examples folder.

8.2.5 Notes on Class `AccountTest`

`static` Methods

A `static` method can call other `static` methods of the same class directly, using the method name by itself, and can manipulate `static` variables in the same class directly. To access the class's instance variables and instance methods, a `static` method must receive a reference to an object of the class as an argument. Instance methods can access all of their class's fields (`static` and instance variables) and methods.

Recall that `static` methods relate to a class as a whole. In contrast, instance methods are associated with a specific object (instance) of the class and may manipulate that object's instance variables. Many objects of a class may exist simultaneously, each with its own copy of the instance variables. Suppose a `static` method were able to invoke an instance method directly. How would the `static` method know which object's instance variables to manipulate? What would happen if no objects of the class existed at the time the instance method was invoked? Thus, Java does not allow a `static` method to directly access instance variables and instance methods of the same class.

`import` Declarations

The `import` declaration in Fig. 8.2 (line 3) indicates to the compiler that the program uses class `Scanner`. In Chapter 2, we discussed that classes `System` and `String` are in package `java.lang`, which is implicitly imported into every Java program, so all programs can use its classes without explicitly importing them. You must import most other classes explicitly.

There's a special relationship between classes in the same folder, like `Account` and `AccountTest`. By default, such classes are in the same **default package**. Classes in the same package are implicitly imported into one another's source-code files. So, an `import` declaration is not required when a class uses another in the same package, as in Fig. 8.2.

Fully Qualifying Class Names with Package Names

The `import` declaration in line 3 of Fig. 8.2 is not required if we always refer to `Scanner` using its **fully qualified class name** `java.util.Scanner`, which includes the full package name and class name. For example, line 8 of Fig. 8.2 could be written as

Click here to view code image

```
var input = new java.util.Scanner(System.in);
```

The Java compiler does not require `import` declarations if the fully qualified class name is specified every time a class name is used. Most Java programmers prefer the more concise programming style enabled by `import` declarations.

Generative AI

1 Prompt genAIs to explain why Java `static` methods cannot directly access instance variables and methods of the same class.

8.2.6 Software Engineering with `private` Instance Variables and `public` *set* and *get* Methods

As you'll see, via *set* and *get* methods, you can validate attempted modifications to `private` data and control how that data is presented to the caller—these are compelling software engineering benefits. We'll discuss this in more detail in subsequent examples.

If the instance variable were `public`, any **client of the class**—that is, any other class that calls the class's methods—could access the data and do whatever it wanted with it, including setting it to an invalid value.

You might think that even though a client of the class cannot directly access a `private` instance variable, the client can do whatever it wants with the variable through `public` *set* and *get* methods. You'd think that you could peek at the `private` data any time with the `public` *get* method and modify the `private` data at will through the `public` *set* method. But you can program *set* methods to validate their arguments and reject any attempts to *set* the data to invalid values, such as a negative body temperature, a day in March out of the range 1 through 31, a product code not in the company's product catalog, etc. Similarly, a *get* method can present the data in a different form. For example, a `Grade` class might store a grade as an `int` between 0 and 100, but a `getGrade` method might return a letter grade as a `String`, such as `"A"` for grades between 90 and 100, `"B"` for grades between 80 and 89, etc. Tightly controlling the access to and presentation of `private` data can reduce errors, making your programs more robust and secure.

Declaring instance variables with access modifier `private` is

known as information hiding. When a program creates (instantiates) an `Account` object, the `private` variable `name` is hidden and can be accessed only by methods of the object's class. You should declare each instance variable and method with an access modifier. Generally, instance variables should be `private` and methods `public`. Later in the book, we'll discuss why you might want to declare a method `private`.

Conceptual View of an `Account` Object with Encapsulated Data

You can think of an `Account` object as shown in the following (non-UML) diagram:

The `private` instance variable `name` is hidden inside the object, represented by the inner circle containing `name`, and accessed via the protective outer layer of `public` methods, represented by the outer circle containing `getName` and `setName`. Client code that needs to interact with the `Account` object's `name` can do so only by calling its `public` methods.

8.3 Default and Explicit Instance Variable Initialization

Instance variables of types `byte`, `char`, `short`, `int`, `long`, `float` and `double` are initialized to 0, and `boolean`s to `false`. You can specify an initial value for a primitive-type instance variable in its declaration, as in

Click here to view code image

```
private int numberOfStudents = 10;
```

If not explicitly initialized, reference-type instance variables are initialized to `null`, representing a "reference to nothing." That's why the first `getName` call in line 14 of Fig. 8.2 returned `null`—`name` was not set yet, so its default initial value was returned.

Generative AI

1 Prompt genAIs to provide insights as to why Java instance variables are initialized by default but Java local variables are not.

8.4 `Account` Class: Initializing Objects with Constructors

You saw in Section 8.2 that an `Account` object's `name` instance variable was initialized to `null` by default. What if you want to provide a name when you create an `Account` object? Each class you declare can optionally provide a constructor with parameters whose values are used to initialize an object's instance variables when it's created. Java requires a constructor call for every new object, so this is the ideal point to initialize an object's instance variables. The next example enhances the `Account` class with a constructor that receives a name and uses it to initialize a new `Account` object's `name`.

8.4.1 Declaring an `Account` Constructor for Custom Object Initialization

You might want to specify a name for an `Account` object when the object is created, as you'll see in line 8 of Fig. 8.4:

Click here to view code image

```
var account1 = new Account("Wei Li");
```

The `String` argument `"Wei Li"` is passed to the `Account` object's constructor and used to initialize the `name` instance variable. The preceding statement requires that the class provide a constructor that takes only a `String` parameter. Figure 8.3 contains a modified `Account` class with such a constructor.

Click here to view code image

```
1  // Fig. 8.3: Account.java
2  // Account class with a constructor that initializes the name.
3
4  public class Account {
5     private String name; // instance variable
6
7     // constructor initializes name with parameter name
8     public Account(String name) { // constructor name is class name
9        this.name = name;
10    }
11
12    // method to set the name
13    public void setName(String name) {
```

```
14          this.name = name;
15       }
16
17       // method to retrieve the name
18       public String getName() {
19          return name;
20       }
21    }
```

Fig. 8.3 | `Account` class with a constructor that initializes the `name`.

`Account` Constructor Declaration

Lines 8–10 of Fig. 8.3 declare `Account`'s constructor, which must have the same name as the class. Line 8 indicates that this constructor has one parameter—a `String` called `name`. When you create a new `Account` object, you'll pass a person's name to the `name` parameter, which the constructor will assign to instance variable `name` (line 9). You should not call a class's non-`static` methods from its constructors. We'll discuss why this is problematic in Chapter 9. Section 8.16 discusses `static` class methods.

Generative AI

1 Prompt genAIs, asking whether Java constructors are methods.
2 Prompt genAIs to explain—without mentioning inheritance or polymorphism (discussed in Chapter 9)—why you should not call a Java class's methods from its constructors.

8.4.2 Class `AccountTest`: Initializing New `Account` Objects

The `AccountTest` program (Fig. 8.4) initializes two `Account` objects using the constructor. Line 8 creates and initializes the new `Account` object `account1`, passing the argument `"Wei Li"` to the constructor to initialize that object's `name`. Line 9 repeats this process, passing the argument `"Logan Brown"` to initialize `account2`'s `name`. Lines 12–13 use each object's `getName` method to obtain its `name`. The output shows different names, confirming that each `Account` maintains its own copy of the instance variable `name`.

Click here to view code image

```
1  // Fig. 8.4: AccountTest.java
2  // Using the Account constructor to initialize the name
3  // instance variable of each new Account object.
4
5  public class AccountTest {
6     public static void main(String[] args) {
7        // create two Account objects
8        var account1 = new Account("Wei Li");
9        var account2 = new Account("Logan Brown");
10
11       // display initial value of name for each Account
12       System.out.printf("account1 name is: %s%n", account1.getName());
13       System.out.printf("account2 name is: %s%n", account2.getName());
14    }
15 }
```

```
account1 name is: Wei Li
account2 name is: Logan Brown
```

Fig. 8.4 | Using the `Account` constructor to initialize the `name` instance variable of each new `Account` object.

Constructors Cannot Return Values

A difference between constructors and methods is that constructors cannot return values, so they do not specify a return type (not even `void`). Usually, constructors are `public` so other classes can call them—in Chapter 9, we'll discuss when to use `private` constructors.

Default Constructor

Recall that line 11 of Fig. 8.2

Click here to view code image

```
var myAccount = new Account();
```

used `new` to create an `Account` object. The empty parentheses after "`new Account`" indicate a call to the **class's default constructor**. The compiler provides a default constructor with no parameters in any class that does not explicitly declare a constructor. When a class has only the default constructor, its instance variables are initialized to their default values. In Section 8.10, you'll see that classes can have multiple constructors.

There's No Default Constructor in a Class That Declares a Constructor

If you declare a constructor with one or more parameters, you will not be able to create an `Account` with the expression `new Account()` as in Fig. 8.2, unless you also declare a custom constructor with no parameters. If default initialization of your class's instance variables is not acceptable, provide a custom constructor to ensure each new object's instance variables are initialized with meaningful values.

Adding the Constructor to Class `Account`'s UML Class Diagram

The following is the UML class diagram for class `Account` (Fig. 8.3), which has a constructor with a `String name` parameter:

The UML models a constructor with other operations in a class diagram's third compartment and distinguishes it from the class's operations by enclosing the word "constructor" in **guillemets (« and »)** before its name. It's customary to list constructors before other operations in the third compartment and before other operations in the class declaration.

Generative AI

1 Prompt genAIs to provide insights as to why Java constructors cannot return values.

8.5 `Account` Class with a Balance

We now declare an `Account` class that maintains a bank account's name and balance. Class `Account` represents the account balance as a `BigDecimal`.

8.5.1 `Account` Class with a `BigDecimal` `balance` Instance Variable

Our next version of class `Account` (Fig. 8.5) maintains `name` and `balance` instance variables. A typical bank services many accounts, each with its own balance, so line 8 declares an instance variable `balance` of type `BigDecimal`.

Click here to view code image

```
1  // Fig. 8.5: Account.java
2  // Account class with a BigDecimal instance variable balance, and
3  // a constructor and deposit method that perform validation.
4  import java.math.BigDecimal;
5
6  public class Account {
7     private String name; // instance variable
8     private BigDecimal balance = BigDecimal.ZERO; // instance variable
9
10    // Account constructor that receives two parameters
11    public Account(String name, BigDecimal balance) {
12       this.name = name; // assign name to instance variable name
13
14       // validate that the balance is greater than 0; if it's not,
15       // instance variable balance keeps its default initial value of 0
16       if (balance.compareTo(BigDecimal.ZERO) > 0) { // is balance valid?
17          this.balance = balance; // assign it to instance variable balance
18       }
19    }
20
21    // method that deposits (adds) only a valid amount to the balance
22    public void deposit(BigDecimal amount) {
23       if (amount.compareTo(BigDecimal.ZERO) > 0) { // is amount valid?
24          balance = balance.add(amount); // add it to the balance
25       }
26    }
27
28    // method returns the account balance
29    public BigDecimal getBalance() {
30       return balance;
31    }
32
33    // method that sets the name
34    public void setName(String name) {
35       this.name = name;
36    }
37
38    // method that returns the name
39    public String getName() {
40       return name;
41    }
42 }
```

Fig. 8.5 | `Account` class with a `BigDecimal` instance variable `balance` and a constructor and `deposit` method that perform validation.

`Account` Class Two-Parameter Constructor

The class has a constructor and four methods. It's common for someone opening an account to deposit money immediately, so the constructor (lines 11–19) now receives a second parameter—`BigDecimal balance`—representing the starting balance. Lines 16–18 ensure that the `balance` parameter's value is greater than `BigDecimal.ZERO`. If so, line 17 assigns it to the instance variable `balance`. Otherwise, the instance variable `balance` remains at `BigDecimal.ZERO`—its default initial value in line 8. Later in this chapter, we'll throw exceptions for invalid initial values.

`Account` Class `deposit` Method

Method `deposit` (lines 22–26) does not return any data when it completes execution, so its return type is `void`. The method receives a `BigDecimal` parameter named `amount` that we add to the instance variable `balance` only if the parameter's value is greater than the value `BigDecimal.ZERO`; otherwise, `balance` remains unchanged.

`Account` Class `getBalance` Method

Method `getBalance` (lines 29–31) specifies no parameters and returns a `BigDecimal`, enabling the class's clients to obtain a particular `Account` object's `balance`.

Generative AI

1 Prompt genAIs asking why Java does not allow you to use convenient operators like `+` and `-` when manipulating `BigDecimal` monetary amounts, as you can with primitive types like `int` and `double`.

8.5.2 `AccountTest` Class

Class `AccountTest` (Fig. 8.6) creates two `Account` objects (lines 9–10) and initializes them with a valid balance of `50.00` and an invalid balance of `-7.53`, respectively—for our examples, we assume that balances must be greater than or equal to zero. The calls to method `printf` in lines 13–16 display the account names and balances, obtained by calling each `Account`'s `getName` and `getBalance` methods. For the balances, we call each `BigDecimal` object's `setScale` method to format the balance with two digits to the right of the decimal point using `RoundingMode.HALF_EVEN`, as introduced in Section 4.11.

Click here to view code image

```
1  // Fig. 8.6: AccountTest.java
2  // Inputting and outputting floating-point numbers with Account objects.
3  import java.math.BigDecimal;
4  import java.math.RoundingMode;
5  import java.util.Scanner;
6
7  public class AccountTest {
8     public static void main(String[] args) {
9        var account1 = new Account("Wei Li", new BigDecimal("50.00"));
10       var account2 = new Account("Logan Brown", new BigDecimal("-7.53"));
11
12       // display initial balance of each object
13       System.out.printf("%s balance: $%s%n", account1.getName(),
14          account1.getBalance().setScale(2, RoundingMode.HALF_EVEN));
15       System.out.printf("%s balance: $%s%n%n", account2.getName(),
16          account2.getBalance().setScale(2, RoundingMode.HALF_EVEN));
17
18       // create a Scanner to obtain input from the user
19       var input = new Scanner(System.in);
20
21       System.out.print("Enter deposit amount for account1: "); // prompt
22       var depositAmount = input.nextBigDecimal(); // get input
23       System.out.printf("%nadding %s to account1 balance%n%n",
24          depositAmount.setScale(2, RoundingMode.HALF_EVEN));
25       account1.deposit(depositAmount); // add to account1 balance
26
27       // display balances
28       System.out.printf("%s balance: $%s%n", account1.getName(),
29          account1.getBalance().setScale(2, RoundingMode.HALF_EVEN));
30       System.out.printf("%s balance: $%s%n%n", account2.getName(),
31          account2.getBalance().setScale(2, RoundingMode.HALF_EVEN));
32
33       System.out.print("Enter deposit amount for account2: "); // prompt
34       depositAmount = input.nextBigDecimal(); // get input
35       System.out.printf("%nadding %s to account2 balance%n%n",
36          depositAmount.setScale(2, RoundingMode.HALF_EVEN));
37       account2.deposit(depositAmount); // add to account2 balance
38
39       // display balances
40       System.out.printf("%s balance: $%s%n", account1.getName(),
41          account1.getBalance().setScale(2, RoundingMode.HALF_EVEN));
42       System.out.printf("%s balance: $%s%n%n", account2.getName(),
43          account2.getBalance().setScale(2, RoundingMode.HALF_EVEN));
44    }
45 }
```

```
Wei Li balance: $50.00
```

```
Logan Brown balance: $0.00

Enter deposit amount for account1: 25.53

adding 25.53 to account1 balance

Wei Li balance: $75.53
Logan Brown balance: $0.00

Enter deposit amount for account2: 123.45

adding 123.45 to account2 balance

Wei Li balance: $75.53
Logan Brown balance: $123.45
```

Fig. 8.6 | Inputting and outputting floating-point numbers with `Account` objects.

Displaying the `Account` Objects' Initial Balances

When we display the initial balances in lines 13–16, `account2`'s `balance` is `0.00` because the constructor rejected the attempt to start `account2` with a negative balance. So that object's `balance` retained its default initial value.

Reading a `BigDecimal` Value from the User and Making a Deposit

Line 21 prompts the user to enter a deposit amount for `account1`. Line 22 declares the local variable `depositAmount` to store each deposit amount the user enters. Line 22 also obtains the user input by calling the **`Scanner`'s `nextBigDecimal` method**, which converts the input to a `BigDecimal` and returns it. Lines 23–24 display the `depositAmount`. Then, line 25 calls object `account1`'s `deposit` method with the `depositAmount` as the argument. Method `deposit` ensures that the `depositAmount` is valid and, if so, adds it to the `balance`. Lines 28–31 output the `name`s and `balance`s of both `Account`s again to show that only `account1`'s `balance` has changed.

Lines 33–36 prompt for and input `account2`'s deposit amount, then display it. Line 37 calls object `account2`'s `deposit` method with `depositAmount` as the argument to add that value to the `balance`. Finally, lines 40–43 output the `name`s and `balance`s of both `Account`s again to show that only `account2`'s `balance` has changed.

Duplicated Code in Method `main`

The six statements in lines 13–14, 15–16, 28–29, 30–31, 40–41 and 42–43 are almost identical—they each output an `Account`'s `name` and `balance`. They differ only in the name of the `Account` object—`account1` or `account2`. Similarly, lines 21–25 and 33–37 are almost identical for inputting a deposit amount and depositing it into each `Account`. Duplicate code like this can create code maintenance problems when that code needs to be updated. Replacing duplicated code with calls to a method that contains one copy of that code can reduce the size of your program and improve its maintainability.

UML Class Diagram for Class `Account`

The following is the UML class diagram for Fig. 8.5's `Account` class. The second compartment models the `private` attributes `name` (a `String`) and `balance` (a `BigDecimal`). The third compartment models the constructor and its `name` (a `String`) and `balance` (a `BigDecimal`) parameters. The third compartment also models the class's `public` methods—`deposit` with a `BigDecimal amount` parameter, `getBalance` with a `BigDecimal` return type, `setName` with a `String name` parameter and `getName` with a `String` return type.

Generative AI

1 Prompt genAIs to explain "DRY" (don't repeat yourself) coding to help you avoid duplicate code.
2 Prompt genAIs with the code in Fig. 8.6 asking them to refactor the program to eliminate the code duplication in `main`.

8.6 Case Study: Card Shuffling and Dealing Simulation

Chapter 6's examples demonstrated arrays containing only primitive-type elements. An array's elements can have primitive or reference types. This section uses random-number generation and an array of reference-type elements—namely objects representing playing cards—to develop a class that simulates card shuffling and

dealing, which you can use to implement programs that play card games.

This example contains three classes. The `Card` class (Fig. 8.7) represents a playing card with a face (`"Ace"`, `"Deuce"`, `"Three"`, ..., `"Jack"`, `"Queen"`, `"King"`) and a suit (`"Hearts"`, `"Diamonds"`, `"Clubs"`, `"Spades"`). The `DeckOfCards` class (Fig. 8.8) creates a deck of 52 playing cards in which each element is a `Card` object. Finally, the `DeckOfCardsTest` class (Fig. 8.9) demonstrates class `DeckOfCards`'s card-shuffling-and-dealing capabilities.

Class `Card`

Class `Card` (Fig. 8.7) contains `String` instance variables `face` and `suit` that store references to a specific `Card`'s face and suit names. We declared these `final` because a real card's face and suit cannot be modified.[4] The `Card` constructor (lines 9–12) receives two `String`s to initialize `face` and `suit`. Method `toString` (lines 15–17) returns a `String` consisting of the `Card`'s `face`, `"of "` and the `Card`'s `suit` (e.g., `"Ace of Spades"`).[5] Java calls `toString` implicitly when an object is used where a `String` is expected—for example, when `printf` outputs an object as a `String` using the `%s` format specifier or when you concatenate an object with a `String` using the `+` operator. For this behavior to occur, you must declare `toString` with the header shown in line 15.

4. We'll discuss `final` instance variables in more detail in Section 8.18.

5. You'll learn in Chapter 9 that when we provide a custom `toString` method, we are "overriding" a version "inherited" from class `Object`. As of Chapter 9, every method we explicitly override will be preceded by the annotation `@Override`, which prevents a common programming error.

Click here to view code image

```
1  // Fig. 8.7: Card.java
2  // Card class represents a playing card.
3
4  public class Card {
5     private final String face; // face of card ("Ace", "Deuce", ...)
6     private final String suit; // suit of card ("Hearts", "Diamonds", ...)
7
8     // two-argument constructor initializes card's face and suit
9     public Card(String face, String suit) {
10       this.face = face; // initialize face of card
11       this.suit = suit; // initialize suit of card
12    }
```

```
13
14      // return String representation of Card
15      public String toString() {
16         return face + " of " + suit;
17      }
18   }
```

Fig. 8.7 | `Card` class represents a playing card.

Class `DeckOfCards`

Class `DeckOfCards` (Fig. 8.8) creates and manages an array of `Card` references. We split this class into parts for discussion purposes. `NUMBER_OF_CARDS` (line 9) specifies the number of `Card`s in a deck (52). Line 10 declares and initializes the instance variable `deck`, which refers to a new array of `Card`s with `NUMBER_OF_CARDS` (52) elements. The `deck` array's elements are `null` by default. Recall that `null` represents a "reference to nothing," so no `Card` objects exist yet. Class `DeckOfCards` also declares `int` instance variable `currentCardIndex` (line 11), representing the sequence number (0–51) of the next `Card` to deal from the `deck` array.

Click here to view code image

```
1   // Fig. 8.8: DeckOfCards.java
2   // DeckOfCards class represents a deck of playing cards.
3   import java.util.random.RandomGenerator;
4
5   public class DeckOfCards {
6      // random number generator
7      private static final RandomGenerator randomNumbers =
8         RandomGenerator.getDefault();
9      public static final int NUMBER_OF_CARDS = 52; // constant # of Cards
10     private Card[] deck = new Card[NUMBER_OF_CARDS]; // Card references
11     private int currentCardIndex = 0; // index of next Card to deal (0-51)
12
```

Fig. 8.8 | `DeckOfCards` class represents a deck of playing cards.

`DeckOfCards` Constructor

The constructor uses a loop (lines 20–22) to fill the `deck` array with `Card` objects. The loop iterates from `0` while the `count` is less than `deck.length`, causing `count` to take on each value from 0 through 51 (the array's indices). We initialize each `Card` with two `String`s—one from the `faces` array containing the `String`s `"Ace"` through

`"King"` and one from the `suits` array containing the `String`s `"Hearts"`, `"Diamonds"`, `"Clubs"` and `"Spades"`. The calculation `count % 13` results in a value from 0 to 12 (the 13 indices of the `faces` array in lines 15–16), and the calculation `count / 13` results in a value from 0 to 3 (the four indices of the `suits` array in line 17). When the loop completes, the `deck` contains `Card`s with faces `"Ace"` through `"King"` of each suit—13 `"Hearts"`, then 13 `"Diamonds"`, then 13 `"Clubs"`, then 13 `"Spades"`. This example represents the faces and suits as arrays of `String`s. You could also represent the faces and suits as arrays of `enum` constants.

Click here to view code image

```
13      // constructor fills deck of Cards
14      public DeckOfCards() {
15         String[] faces = {"Ace", "Deuce", "Three", "Four", "Five", "Six",
16            "Seven", "Eight", "Nine", "Ten", "Jack", "Queen", "King"};
17         String[] suits = {"Hearts", "Diamonds", "Clubs", "Spades"};
18
19         // populate deck with Card objects
20         for (int count = 0; count < deck.length; ++count) {
21            deck[count] = new Card(faces[count % 13], suits[count / 13]);
22         }
23      }
24
```

`DeckOfCards` Method `shuffle`

Method `shuffle` (lines 26–40) shuffles the `Card`s. For each `Card`, line 33 chooses a random index 0–51 to select another `Card`. Then, lines 36–38 swap the current and randomly chosen `Card`s in the array. The extra variable `temp` (line 36) temporarily stores one of the two `Card` objects being swapped. The `for` loop shuffles the `Card` objects by performing 52 swaps in a single pass of the array.

Click here to view code image

```
25      // shuffle deck of Cards with one-pass algorithm
26      public void shuffle() {
27         // next call to method dealCard should start at deck[0] again
28         currentCardIndex = 0;
29
30         // for each Card, pick another random Card (0-51) and swap them
31         for (int first = 0; first < deck.length; ++first) {
32            // select a random number between 0 and 51
33            int second = randomNumbers.nextInt(NUMBER_OF_CARDS);
34
```

```
35            // swap current Card with randomly selected Card
36            Card temp = deck[first];
37            deck[first] = deck[second];
38            deck[second] = temp;
39         }
40      }
41
```

Method `shuffle`'s algorithm is considered biased because it sometimes swaps a `Card` with itself. You should use an unbiased algorithm for real games to ensure that all possible shuffled sequences are equally likely to occur. Refer to this section's genAI exercises, where you'll investigate how the Fisher-Yates unbiased shuffling algorithm works and ask the generative AIs to modify the `shuffle` method to use the unbiased algorithm.

Why Do We Need Variable `temp` to Perform the Swap?

You cannot perform the swap in lines 36–38 with only the two statements

```
deck[first] = deck[second];
deck[second] = deck[first];
```

If `deck[first]` is the `"Ace"` of `"Spades"` and `deck[second]` is the `"Queen"` of `"Hearts"`, after the first assignment, both array elements would contain the `"Queen"` of `"Hearts"` and the `"Ace"` of `"Spades"` would be lost.

`DeckOfCards` Method `dealCard`

Method `dealCard` (lines 43–51) deals one `Card`. Recall that `currentCardIndex` is the index of the next `Card` to deal—the `Card` at the deck's top. Line 45 compares `currentCardIndex` to the `deck` array's `length`. If the `deck` is not empty (`currentCardIndex` is less than 52), line 46 returns the top `Card` and postincrements `currentCardIndex` to prepare for the next `dealCard` call; otherwise, line 49 returns `null`.

Click here to view code image

```
42      // deal one Card
43      public Card dealCard() {
44         // determine whether Cards remain to be dealt
45         if (currentCardIndex < deck.length) {
```

```
46             return deck[currentCardIndex++]; // return current Card in array
47          }
48          else {
49             return null; // return null to indicate that all Cards were dealt
50          }
51       }
52    } // end class DeckOfCards
```

Shuffling and Dealing Cards

The program of Fig. 8.9 demonstrates the `DeckOfCards` class. Line 7 creates a `DeckOf-Cards` object named `myDeckOfCards`, calling its constructor to create the deck with the 52 `Card` objects in order by suit and face. Line 8 invokes `myDeckOfCards`'s `shuffle` method to rearrange the `Card` objects. Lines 11–18 deal all 52 `Card`s, printing them in four columns of 13 `Card`s each. Line 13 deals one `Card` object by invoking `myDeckOfCards`'s `deal-Card` method, then displays the `Card` left justified in a field of 19 characters. When a `Card` is output as a `String`, Java invokes its `toString` method (Fig. 8.7) implicitly. Lines 15–17 in Fig. 8.9 start a new line after every four `Card`s.

Click here to view code image

```
1  // Fig. 8.9: DeckOfCardsTest.java
2  // Card shuffling and dealing.
3
4  public class DeckOfCardsTest {
5     // execute program
6     public static void main(String[] args) {
7        var myDeckOfCards = new DeckOfCards();
8        myDeckOfCards.shuffle(); // place Cards in random order
9
10       // print all 52 Cards in the order in which they are dealt
11       for (int i = 1; i <= DeckOfCards.NUMBER_OF_CARDS; ++i) {
12          // deal and display a Card
13          System.out.printf("%-19s", myDeckOfCards.dealCard());
14
15          if (i % 4 == 0) { // output a newline after every fourth card
16             System.out.println();
17          }
18       }
19    }
20 }
```

```
Six of Spades       Eight of Spades     Six of Clubs        Nine of Hearts
Queen of Hearts     Seven of Clubs      Nine of Spades      King of Hearts
```

```
Three of Diamonds   Deuce of Clubs      Ace of Hearts       Ten of Spades
Four of Spades      Ace of Clubs        Seven of Diamonds   Four of Hearts
Three of Clubs      Deuce of Hearts     Five of Spades      Jack of Diamonds
King of Clubs       Ten of Hearts       Three of Hearts     Six of Diamonds
Queen of Clubs      Eight of Diamonds   Deuce of Diamonds   Ten of Diamonds
Three of Spades     King of Diamonds    Nine of Clubs       Six of Hearts
Ace of Spades       Four of Diamonds    Seven of Hearts     Eight of Clubs
Deuce of Spades     Eight of Hearts     Five of Hearts      Queen of Spades
Jack of Hearts      Seven of Spades     Four of Clubs       Nine of Diamonds
Ace of Diamonds     Queen of Diamonds   Five of Clubs       King of Spades
Five of Diamonds    Ten of Clubs        Jack of Spades      Jack of Clubs
```

Fig. 8.9 | Card shuffling and dealing.

Preventing `NullPointerExceptionS`

In Fig. 8.8, we created a `deck` array of 52 `Card` references—by default, each is initialized to `null`. A `NullPointerException` occurs when you try to call a method on a `null` reference. In industrial-strength code, ensuring references are not `null` before you use them to call methods prevents `NullPointerException`s.

Generative AI

1 Prompt genAIs, asking for cases in which Java may throw `NullPointerException`s and how the improved `NullPointerException` error messages introduced by Java Enhancement Proposal (JEP) 358 help you locate code that causes them.

2 Prompt genAIs with the `shuffle` method in Fig. 8.8, asking them to explain what it means for a shuffling algorithm to be biased and why `shuffle`'s algorithm is biased.

3 Prompt genAIs asking how the unbiased Fisher-Yates shuffling algorithm works and have them reimplement the `DeckOfCards` class's `shuffle` method using it. Test the updated code.

8.7 Case Study: `Time` Class

Our next example consists of classes `Time1` (Fig. 8.10) and `Time1Test` (Fig. 8.11). Class `Time1` represents the time of day. Class

`Time1Test`'s `main` method creates one `Time1` object and invokes its methods. The program's output appears in Fig. 8.11.

`Time1` Class Declaration

Class `Time1`'s `private int` instance variables `hour`, `minute` and `second` (Fig. 8.10, lines 5–7) represent the time in universal-time format (24-hour clock format in which hours are in the range 0–23, and minutes and seconds are each in the range 0–59). `Time1` contains `public` methods `setTime` (lines 11–22), `toUniversalString` (lines 25–27) and `toString` (lines 30–34). These methods are the **`public` services** or the **`public` interface** the class provides to its clients.

Click here to view code image

```
1  // Fig. 8.10: Time1.java
2  // Time1 class declaration maintains the time in 24-hour format.
3
4  public class Time1 {
5     private int hour; // 0 - 23
6     private int minute; // 0 - 59
7     private int second; // 0 - 59
8
9     // set a new time value using universal time; throw an
10    // exception if the hour, minute or second is invalid
11    public void setTime(int hour, int minute, int second) {
12       // validate hour, minute and second
13       if (hour < 0 || hour >= 24 || minute < 0 || minute >= 60 ||
14          second < 0 || second >= 60) {
15          throw new IllegalArgumentException(
16             "hour, minute and/or second was out of range");
17       }
18
19       this.hour = hour;
20       this.minute = minute;
21       this.second = second;
22    }
23
24    // convert to String in universal-time format (HH:MM:SS)
25    public String toUniversalString() {
26       return String.format("%02d:%02d:%02d", hour, minute, second);
27    }
28
29    // convert to String in standard-time format (H:MM:SS AM or PM)
30    public String toString() {
31       return String.format("%d:%02d:%02d %s",
32          ((hour == 0 || hour == 12) ? 12 : hour % 12),
```

```
33             minute, second, (hour < 12 ? "AM" : "PM"));
34      }
35   }
```

Fig. 8.10 | `Time1` class declaration maintains the time in 24-hour format.

Default Constructor

Class `Time1` does not declare a constructor, so the compiler supplies a default constructor. Each instance variable implicitly receives the default `int` value (`0`).

Method `setTime` and Throwing Exceptions

Method `setTime` (lines 11–22) is a `public` method with three `int` parameters used to set the time. Lines 13–14 test each argument to determine whether the value is outside the proper range. The `hour` value must be greater than or equal to `0` and less than `24` because the universal-time format represents hours as integers from 0 to 23 (e.g., 1 PM is hour 13, 11 PM is hour 23, midnight is hour 0, and noon is hour 12). Similarly, both `minute` and `second` values must be greater than or equal to `0` and less than `60`. If the argument values are valid, lines 19–21 assign them to the `hour`, `minute` and `second` instance variables. You should modify an object's instance variables only if a method's arguments are all valid.

For invalid values, `setTime` throws an **`IllegalArgumentException`** (lines 15–16) to notify the client code of the problem. Line 15 creates a new `IllegalArgumentException` object and calls its constructor that receives a custom error message. Next, the **`throw` statement** immediately terminates `setTime` and returns the exception to the calling method. Then, the client code can handle the exception using `try`...`catch` (Section 6.12).

Method `toUniversalString`

Method `toUniversalString` (lines 25–27) takes no arguments and returns a `String` in universal-time format, consisting of two digits each for the hour, minute and second—recall that you can use the `0` flag in a `printf` format specification (e.g., `"%02d"`) to display leading zeros for a value that doesn't use all the character positions in the specified field width. For example, if the time were 1:30:07 PM, the method would return `13:30:07`. Line 26 uses **`static`**

`String` method `format` to return a `String` containing the formatted `hour`, `minute` and `second` values, each with two digits and possibly a leading `0` (specified with the `0` flag). The `format` method is similar to `printf`, but returns a formatted `String` rather than displaying it.

Method `toString`

Method `toString` (lines 30–34) takes no arguments and returns a `String` in 12-hour format, consisting of the `hour`, `minute` and `second` values separated by colons and followed by AM or PM (e.g., `11:30:17 AM` or `1:27:06 PM`). Like `toUniversalString`, `toString` uses `static String` method `format` to format the `minute` and `second` as two-digit values, with leading zeros if necessary. Line 32 uses a conditional operator (`?:`) to determine the `hour` value in the `String`. If the `hour` is `0` or `12` (AM or PM), it appears as 12; otherwise, it appears as a value from 1 to 11. The conditional operator in line 33 determines whether AM or PM will be returned as part of the `String`.

Using Class `Time1`

Class `Time1Test` (Fig. 8.11) uses class `Time1`. Line 7 declares the `Time1` variable `time` and initializes it with a new `Time1` object. Operator `new` implicitly invokes class `Time1`'s default constructor because `Time1` does not declare any constructors. To confirm that the `Time1` object was initialized properly, line 10 calls the `private` method `displayTime` (lines 31–34), which, in turn, calls the `Time1` object's `toUniversalString` and `toString` methods to output the time in universal-time format and standard-time format, respectively. Note that `toString` could have been called implicitly here rather than explicitly. Next, line 14 calls the `time` object's `setTime` method to change the time. Then, line 15 outputs the time in both formats again to confirm the time was set correctly.

Click here to view code image

```
1  // Fig. 8.11: Time1Test.java
2  // Time1 object used in an app.
3
4  public class Time1Test {
5     public static void main(String[] args) {
6        // create and initialize a Time1 object
7        var time = new Time1(); // invokes Time1 constructor
8
```

```
9          // output string representations of the time
10         displayTime("After time object is created", time);
11         System.out.println();
12
13         // change time and output updated time
14         time.setTime(13, 27, 6);
15         displayTime("After calling setTime", time);
16         System.out.println();
17
18         // attempt to set time with invalid values
19         try {
20            time.setTime(99, 99, 99); // all values out of range
21         }
22         catch (IllegalArgumentException e) {
23            System.out.printf("Exception: %s%n%n", e.getMessage());
24         }
25
26         // display time after attempt to set invalid values
27         displayTime("After calling setTime with invalid values", time);
28      }
29
30      // displays a Time1 object in 24-hour and 12-hour formats
31      private static void displayTime(String header, Time1 t) {
32         System.out.printf("%s%nUniversal time: %s%nStandard time: %s%n",
33            header, t.toUniversalString(), t.toString());
34      }
35   }
```

```
After time object is created
Universal time: 00:00:00
Standard time: 12:00:00 AM

After calling setTime
Universal time: 13:27:06
Standard time: 1:27:06 PM

Exception: hour, minute and/or second was out of range

After calling setTime with invalid values
Universal time: 13:27:06
Standard time: 1:27:06 PM
```

Fig. 8.11 | `Time1` object used in an app.

Methods declared `private`, such as `displayTime`, can be called only by other methods of the same class. Such methods are **utility methods** or **helper methods** because that support the operation of the class's other methods.

Calling `Time1` Method `setTime` with Invalid Values

To illustrate that method `setTime` validates its arguments, line 20 calls method `setTime` with invalid arguments of `99` for the `hour`, `minute` and `second`. This statement is placed in a `try` block (lines 19–21) in case `setTime` throws an `IllegalArgumentException`, which it will do since the arguments are all invalid. When this occurs, the exception is caught at lines 22–24, and line 23 displays the exception's error message by calling its `getMessage` method. Line 27 outputs the time again in both formats to confirm that `setTime` did not change the time when invalid arguments were supplied.

Software Engineering of the `Time1` Class Declaration

Consider class `Time1`'s design. Instance variables `hour`, `minute` and `second` are `private`. The data representation used in the class is of no concern to the class's clients. For example, it would be reasonable for `Time1` to represent the time internally as the number of seconds since midnight or the number of minutes and seconds since midnight. Clients could use the same `public` methods and get the same results without being aware of this change. You'll ask generative AIs to make this change in Generative AI question 1.

Classes simplify programming because the client can use only a class's `public` methods. Such methods are client-oriented rather than implementation-oriented. Clients are neither aware of nor involved in a class's implementation. Clients generally care about what the class does, not how the class does it.

Interfaces change less frequently than implementations. When an implementation changes, implementation-dependent code must change accordingly. Hiding the implementation reduces the possibility that other program parts will become dependent on a class's implementation details.

Java Date/Time API

This section's example and several of this chapter's later examples demonstrate various class-implementation concepts in classes that represent dates and times. In professional Java development, you should use the Java Date/Time API's features, like those we introduced in Section 5.12, rather than building your own date and time classes. We use some additional Date/Time API features in Chapter 18.

Generative AI

1 Prompt genAIs with class `Time1` (Fig. 8.10), asking them to refactor it to use one `int` representing the number of seconds since midnight. Tell the generative AIs to ensure `Time1` offers the same `public` interface so existing client code does not require modifications. Compare the updated `Time1` class to Fig. 8.10. Use the updated `Time1` class in the program of Fig. 8.11 to ensure that it produces the same results.

8.8 Controlling Access to Members

The access modifiers `public` and `private` control access to a class's variables and methods. In Chapter 9, we'll introduce the additional access modifier `protected`. The primary purpose of `public` methods is to present to the class's clients a view of the class's services—the class's `public` interface.

Figure 8.12 demonstrates that `private` class members are not accessible outside the class. Lines 7–9 attempt to access the `private` instance variables `hour`, `minute` and `second` of the `Time1` object `time`. When this program is compiled, the compiler generates error messages indicating these `private` members are inaccessible. This program assumes that the `Time1` class from Fig. 8.10 is used. An attempt by a method that's not a member of a class to access a `private` member of that class generates a compilation error.

Click here to view code image

```
1  // Fig. 8.12: MemberAccessTest.java
2  // Private members of class Time1 are not accessible.
3  public class MemberAccessTest {
4     public static void main(String[] args) {
5        var time = new Time1(); // create and initialize Time1 object
6
7        time.hour = 7; // error: hour has private access in Time1
8        time.minute = 15; // error: minute has private access in Time1
9        time.second = 30; // error: second has private access in Time1
10    }
11 }
```

```
.\MemberAccessTest.java:7: error: hour has private access in Time1
      time.hour = 7; // error: hour has private access in Time1
          ^
.\MemberAccessTest.java:8: error: minute has private access in Time1
```

```
        time.minute = 15; // error: minute has private access in Time1
            ^
.\MemberAccessTest.java:9: error: second has private access in Time1
        time.second = 30; // error: second has private access in Time1
            ^
3 errors
error: compilation failed
```

Fig. 8.12 | Private members of class `Time1` are not accessible.

Generative AI

1 Prompt genAIs, asking if Java's `private` access modifier makes data absolutely secure.

8.9 Referring to the Current Object's Members with the `this` Reference

Every object can access a reference to itself with keyword `this`. When an instance method is called for a particular object, the method's body implicitly uses "`this.`" to refer to the object's instance variables and other methods, enabling the class's code to know which object to manipulate. As you'll see in Fig. 8.13, you can also use "`this.`" explicitly in an instance method's body. Section 8.10 shows another interesting use of the `this` keyword. Section 8.16 explains why "`this.`" cannot be used in a `static` method.

Figure 8.13 demonstrates implicit and explicit use of the `this` reference. This example is the first in which we declare two classes in one file—class `ThisTest` is declared in lines 4–9, and class `SimpleTime` in lines 12–41. When you compile a source-code (`.java`) file, the compiler produces a separate `.class` file for every class in that file. In this case, two files are produced—`SimpleTime.class` and `ThisTest.class`. The compiler places these files in the same folder. Note also in Fig. 8.13 that only class `ThisTest` is declared `public`. A source-code (`.java`) file may contain only one `public` class—recall that every public class must be declared in a file that has the class's name followed by the `.java` filename extension. Non-`public` classes can be used only by other classes in the same package, and classes compiled into the same folder are in the same package. So, `SimpleTime` can be used only in class `ThisTest`.

Click here to view code image

```
1  // Fig. 8.13: ThisTest.java
2  // this used implicitly and explicitly to refer to members of an object.
3
4  public class ThisTest {
5     public static void main(String[] args) {
6        var time = new SimpleTime(15, 30, 19);
7        System.out.println(time.buildString());
8     }
9  }
10
11 // class SimpleTime demonstrates the "this" reference
12 class SimpleTime {
13    private int hour; // 0-23
14    private int minute; // 0-59
15    private int second; // 0-59
16
17    // if the constructor uses parameter names identical to
18    // instance variable names the "this" reference is
19    // required to distinguish between the names
20    public SimpleTime(int hour, int minute, int second) {
21       this.hour = hour; // set "this" object's hour
22       this.minute = minute; // set "this" object's minute
23       this.second = second; // set "this" object's second
24    }
25
26    // use explicit and implicit "this" to call toUniversalString
27    public String buildString() {
28       return String.format("%24s: %s%n%24s: %s",
29          "this.toUniversalString()", this.toUniversalString(),
30          "toUniversalString()", toUniversalString());
31    }
32
33    // convert to String in universal-time format (HH:MM:SS)
34    public String toUniversalString() {
35       // "this." is not required here to access instance variables,
36       // because this method does not have local variables with same
37       // names as instance variables
38       return String.format("%02d:%02d:%02d",
39          this.hour, this.minute, this.second);
40    }
41 }
```

```
this.toUniversalString(): 15:30:19
     toUniversalString(): 15:30:19
```

Fig. 8.13 | `this` used implicitly and explicitly to refer to members

of an object.

Class `SimpleTime` (lines 12–41) declares three `private` instance variables—`hour`, `minute` and `second` (lines 13–15). The class's constructor (lines 20–24) receives three `int` arguments to initialize a `SimpleTime` object. Once again, we used parameter names for the constructor that are identical to the class's instance-variable names (lines 13–15), so we use "`this.`" to refer to the instance variables in lines 21–23. Most IDEs will issue a warning if you say `x = x;` instead of `this.x = x;`. The statement `x = x;` is often called a no-op (no operation)—in this case, the statement would assign `x` to itself, which does not modify `x`.

Method `buildString` (lines 27–31) returns a `String` created by a statement that uses the `this` reference to call `toUniversalString` explicitly (line 29) and implicitly (line 30). Both lines perform the same operation. You typically will not use `this` explicitly to reference other methods within the current object. Also, line 39 in `toUniversalString` explicitly uses "`this.`" to access each instance variable, though this is not required because the method has no local variables that shadow the class's instance variables.

There's only one copy of each method per class—every object of the class shares the method's code. On the other hand, each object has its own copy of the class's instance variables. The class's non-`static` methods implicitly use `this` to determine the specific object of the class to manipulate.

Class `ThisTest`'s `main` method (lines 5–8) demonstrates class `SimpleTime`. Line 6 creates a `SimpleTime` object and invokes its constructor. Line 7 invokes the object's `buildString` method then displays the results.

Generative AI

1 Prompt genAIs, asking why, when referring to an instance variable `x` from within a Java class, most IDEs will issue a warning (not an error) if you say

```
x = x;
```

8.10 Case Study: `Time` Class Overloaded Constructors

Next, we demonstrate a class with **overloaded constructors**, enabling objects of our class to be initialized various ways. Like overloaded methods (Section 5.7), you overload constructors by providing multiple constructor declarations with different signatures.

Class `Time2` with Overloaded Constructors

The `Time1` class's default constructor in Fig. 8.10 initialized `hour`, `minute` and `second` to their default `0` values (i.e., midnight in universal time). To initialize objects with non-default values, you must provide custom constructors. Class `Time2` (Fig. 8.14) contains five overloaded constructors that enable flexible initialization of `Time2` objects by passing different arguments. We broke this class into parts for discussion purposes. Lines 5–7 declare the class's `hour`, `minute` and `second` instance variables.

Click here to view code image

```
1  // Fig. 8.14: Time2.java
2  // Time2 class declaration with overloaded constructors.
3
4  public class Time2 {
5     private int hour; // 0 - 23
6     private int minute; // 0 - 59
7     private int second; // 0 - 59
8
```

Fig. 8.14 | `Time2` class declaration with overloaded constructors.

Class `Time2`'s Constructors—Calling One Constructor from Another via `this`

Each constructor accomplishes the same basic job—initialize a `Time2` object's instance variables. Portions of that work are identical for each constructor, so we'll share validation and initialization code by having four of them invoke a fifth that, in turn, ensures the `hour` is in the range 0–23, the `minute` and `second` are each in the range 0–59, then initializes them. As with overloaded functions, the compiler invokes the appropriate constructor by matching the number, types and order of a constructor call's argument types with the number, types and order of the parameter types specified in each constructor declaration. Class `Time2` also provides *set* and *get*

methods for each instance variable.

`Time2` Constructor with No Arguments

Lines 11–13 declare a **no-argument constructor**. Once you declare any constructors in a class, the compiler will not provide a default constructor. This no-argument constructor ensures that class `Time2`'s clients can create `Time2` objects with default values.

Click here to view code image

```
9       // Time2 no-argument constructor:
10      // initializes each instance variable to zero
11      public Time2() {
12         this(0, 0, 0); // delegate to three-argument constructor
13      }
14
```

Line 12 uses `this` to invoke the `Time2` constructor that takes three parameters (lines 26–42) with 0 for the `hour`, `minute` and `second`. Using the `this` keyword as shown here must be the constructor's first statement and is a common way to avoid duplicate initialization code. A constructor that entrusts its work to another constructor in the same class is a **delegating constructor**. We use this syntax in four of the five `Time2` constructors to make the class easier to maintain and modify—we'll say more about this momentarily.

`Time2` Constructor with One `int` Argument

Lines 16–18 declare a `Time2` constructor with a single `int` parameter representing the `hour`. We pass this with 0 for the `minute` and `second` to the constructor at lines 26–42.

Click here to view code image

```
15      // Time2 constructor: hour supplied, minute and second defaulted to 0
16      public Time2(int hour) {
17         this(hour, 0, 0); // delegate to three-argument constructor
18      }
19
```

`Time2` Constructor with Two `int` Arguments

Lines 21–23 declare a `Time2` constructor that receives two `int` parameters representing the `hour` and `minute`. We pass these with 0

for the `second` to the constructor at lines 26–42.

Click here to view code image

```
20      // Time2 constructor: hour and minute supplied, second defaulted to 0
21      public Time2(int hour, int minute) {
22         this(hour, minute, 0); // delegate to three-argument constructor
23      }
24
```

`Time2` Constructor with Three `int` Arguments

Lines 26–42 declare the `Time2` constructor that receives three `int` parameters representing the `hour`, `minute` and `second`. This constructor validates and initializes the instance variables.

Click here to view code image

```
25      // Time2 constructor: hour, minute and second supplied
26      public Time2(int hour, int minute, int second) {
27         if (hour < 0 || hour >= 24) {
28            throw new IllegalArgumentException("hour must be 0-23");
29         }
30
31         if (minute < 0 || minute >= 60) {
32            throw new IllegalArgumentException("minute must be 0-59");
33         }
34
35         if (second < 0 || second >= 60) {
36            throw new IllegalArgumentException("second must be 0-59");
37         }
38
39         this.hour = hour;
40         this.minute = minute;
41         this.second = second;
42      }
43
```

`Time2` Constructor with a `Time2` Argument

Lines 45–48 declare a `Time2` constructor that receives a reference to another `Time2` object. This constructor copies the argument object's `hour`, `minute` and `second` values by passing them to the three-argument constructor. (Some languages call this a copy constructor, but this is not a formal Java concept.) The `time.hour`, `time.minute` and `time.second` expressions directly access the argument object's `hour`, `minute` and `second` instance variables—even though they're

`private`. This is allowed due to a special relationship between objects of the same class. When one object has a reference to another of the same class, the first object can access all the second object's data and methods—even `private` ones.

Click here to view code image

```
44      // Time2 constructor: another Time2 object supplied
45      public Time2(Time2 time) {
46         // delegate to three-argument constructor
47         this(time.hour, time.minute, time.second);
48      }
49
```

Class `Time2`'s `setTime` Method

Method `setTime` (lines 53–69) throws an `IllegalArgumentException` (lines 55, 59 and 63) if any of the method's arguments is out of range—this example tests each argument separately to throw exceptions with more precise error messages. If the arguments are valid, lines 66–68 set the instance variables to the argument values.

Click here to view code image

```
50      // Set Methods
51      // set a new time value using universal time;
52      // validate the data
53      public void setTime(int hour, int minute, int second) {
54         if (hour < 0 || hour >= 24) {
55            throw new IllegalArgumentException("hour must be 0-23");
56         }
57
58         if (minute < 0 || minute >= 60) {
59            throw new IllegalArgumentException("minute must be 0-59");
60         }
61
62         if (second < 0 || second >= 60) {
63            throw new IllegalArgumentException("second must be 0-59");
64         }
65
66         this.hour = hour;
67         this.minute = minute;
68         this.second = second;
69      }
70
```

Class `Time2`'s `setHour`, `setMinute` and `setSecond` Methods

Lines 72–96 declare *set* methods for the hour, minute and second. Each validates its argument before setting the corresponding instance variable.

Click here to view code image

```
71      // validate and set hour
72      public void setHour(int hour) {
73         if (hour < 0 || hour >= 24) {
74            throw new IllegalArgumentException("hour must be 0-23");
75         }
76
77         this.hour = hour;
78      }
79
80      // validate and set minute
81      public void setMinute(int minute) {
82         if (minute < 0 || minute >= 60) {
83            throw new IllegalArgumentException("minute must be 0-59");
84         }
85
86         this.minute = minute;
87      }
88
89      // validate and set second
90      public void setSecond(int second) {
91         if (second < 0 || second >= 60) {
92            throw new IllegalArgumentException("second must be 0-59");
93         }
94
95         this.second = second;
96      }
97
```

Class `Time2`'s `getHour`, `getMinute` and `getSecond` Methods

Lines 100–106 declare *get* methods for the hour, minute and second. Each returns the corresponding instance variable.

Click here to view code image

```
98      // Get Methods
99      // get hour value
100      public int getHour() {return hour;}
```

```
101
102       // get minute value
103       public int getMinute() {return minute;}
104
105       // get second value
106       public int getSecond() {return second;}
107
```

Class `Time2`'s `toUniversalString` and `toString` Methods

Lines 109–119 declare the `toUniversalString` and `toString` methods. Each creates a `Time2` object's `String` representation using the same formatting as class `Time1` (Fig. 8.10).

Click here to view code image

```
108       // convert to String in universal-time format (HH:MM:SS)
109       public String toUniversalString() {
110          return String.format("%02d:%02d:%02d",
111             getHour(), getMinute(), getSecond());
112       }
113
114       // convert to String in standard-time format (H:MM:SS AM or PM)
115       public String toString() {
116          return String.format("%d:%02d:%02d %s",
117             ((getHour() == 0 || getHour() == 12) ? 12 : getHour() % 12),
118             getMinute(), getSecond(), (getHour() < 12 ? "AM" : "PM"));
119       }
120    }
```

Notes Regarding Class `Time2`'s *Set* and *Get* Methods and Constructors

Methods `toUniversalString` and `toString` call `Time2`'s *get* methods in line 111 and lines 117–118, respectively. These methods could have accessed `Time2`'s `private` data directly without calling the *get* methods but there's a software-engineering benefit to calling them.

Our internal time representation uses three `int`s, requiring 12 bytes of memory. Consider changing this to the total seconds since midnight—a single `int` requiring only four bytes. If we make this change, only the methods that access `private` data directly need to change. In this case, we'd modify the three-argument constructor, the `setTime` method and the individual *set* and *get* methods for the `hour`, `minute` and `second`. There would be no need to modify

`toUniversalString` and `toString` because they do not access the data directly. Designing the class in this manner reduces the likelihood of programming errors when altering the class's implementation.

Similarly, each `Time2` delegating constructor could have included a copy of the appropriate statements from the three-argument constructor. This might be slightly more efficient because it would eliminate extra constructor calls. However, duplicating statements makes changing the class's internal data representation more difficult.[6] Having the `Time2` delegating constructors call the constructor with three arguments requires making changes only to the three-argument constructor's implementation. Also, the compiler can optimize programs by removing calls to simple methods and replacing them with the expanded code of their declarations—a technique known as **inlining the code**, which improves program performance.

6. In Chapter 9, we'll explain why we chose to duplicate code between class `Time2`'s constructor and `setTime` methods.

Using Class `Time2`'s Overloaded Constructors

Class `Time2Test` (Fig. 8.15) invokes the overloaded `Time2` constructors (lines 6–10 and 21):

Click here to view code image

```
1  // Fig. 8.15: Time2Test.java
2  // Overloaded constructors used to initialize Time2 objects.
3
4  public class Time2Test {
5     public static void main(String[] args) {
6        var t1 = new Time2(); // 00:00:00
7        var t2 = new Time2(2); // 02:00:00
8        var t3 = new Time2(21, 34); // 21:34:00
9        var t4 = new Time2(12, 25, 42); // 12:25:42
10       var t5 = new Time2(t4); // 12:25:42
11
12       System.out.println("Constructed with:");
13       displayTime("t1: all default arguments", t1);
14       displayTime("t2: hour specified; default minute and second", t2);
15       displayTime("t3: hour and minute specified; default second", t3);
16       displayTime("t4: hour, minute and second specified", t4);
17       displayTime("t5: Time2 object t4 specified", t5);
18
19       // attempt to initialize t6 with invalid values
20       try {
```

```
            var t6 = new Time2(27, 74, 99); // invalid values
         }
         catch (IllegalArgumentException e) {
            System.out.printf("%nException while initializing t6: %s%n",
               e.getMessage());
         }
      }

      // displays a Time2 object in 24-hour and 12-hour formats
      private static void displayTime(String header, Time2 t) {
         System.out.printf("%s%n   %s%n   %s%n",
            header, t.toUniversalString(), t.toString());
      }
   }
```

```
Constructed with:
t1: all default arguments
   00:00:00
   12:00:00 AM
t2: hour specified; default minute and second
   02:00:00
   2:00:00 AM
t3: hour and minute specified; default second
   21:34:00
   9:34:00 PM
t4: hour, minute and second specified
   12:25:42
   12:25:42 PM
t5: Time2 object t4 specified
   12:25:42
   12:25:42 PM

Exception while initializing t6: hour must be 0-23
```

Fig. 8.15 | Overloaded constructors used to initialize `Time2` objects.

Line 6 invokes the `Time2` no-argument constructor. Lines 7–10 demonstrate passing arguments to the `Time2` constructors that receive arguments:

- Line 7 invokes the one-argument constructor with the header:

  ```
  public Time2(int hour)
  ```

- Line 8 invokes the two-argument constructor with the header:

  Click here to view code image

```
public Time2(int hour, int minute)
```

- Line 9 invokes the three-argument constructor with the header:

  Click here to view code image

  ```
  public Time2(int hour, int minute, int second)
  ```

- Line 10 invokes the single-argument constructor with the header:

  ```
  public Time2(Time2 time)
  ```

Next, the program displays the `Time2` objects' `String` representations to confirm that they were correctly initialized (lines 13–17).

Line 21 attempts to initialize `t6` by creating a new `Time2` object with three invalid constructor arguments. When the constructor determines that the `hour` argument is out of range, it `throw`s an `IllegalArgumentException`, which we `catch` at line 23 then display its error message, producing the last line of the output.

Generative AI

1 Prompt genAIs with class `Time2` (Fig. 8.14), asking them to rewrite its code to store the time as one `int` representing the number of seconds since midnight. Tell the generative AIs to ensure the class offers the same `public` interface so that the existing client code does not require modifications. Compare the updated `Time2` class to Fig. 8.14. Use the updated `Time2` class in the program of Fig. 8.15 to ensure that it produces the same results.

8.11 Default and No-Argument Constructors

Every class must have at least one constructor. The compiler creates a default constructor with no arguments in a class that does not contain custom constructors. The default constructor initializes the instance variables to the initial values specified in the instance variables' declarations; otherwise, it uses default values (zero for primitive numeric types, `false` for `boolean` values and `null` for references).

The compiler will not create a default constructor if your class declares constructors. In this case, you must declare a no-argument constructor if default initialization is required. Like a default constructor, a no-argument constructor is invoked with empty parentheses. The `Time2` no-argument constructor (lines 11–13 of Fig. 8.14) explicitly initializes a `Time2` object by passing to the three-argument constructor 0 for each parameter. Since 0 is the default value for `int` instance variables, the no-argument constructor in `Time2` could be declared with an empty body. In this case, each instance variable would receive its default value when the no-argument constructor is called. If we omit the no-argument constructor, clients of this class would not be able to create a `Time2` object with the expression `new Time2()`.

Ensure that you do not include a return type in a constructor definition. Java allows other methods of the class besides its constructors to have the same name as the class and specify return types. Such methods are not constructors and will not be called when an object of the class is instantiated.

Generative AI

1 Prompt genAIs for why Java's designers chose to allow other methods of the class besides its constructors to have the same name as the class and specify return types, given this could lead to confusion.

8.12 Notes on *Set* and *Get* Methods

As you know, client code can manipulate a class's `private` fields only through the class's methods. *Set* methods are commonly called **mutator methods** because they change an object's state—i.e., modify the values of instance variables.[7] *Get* methods are commonly called **accessor methods** or **query methods**.

7. Other methods besides *set* methods can also be mutator methods.

Set and Get Methods vs. `public` Data

It would seem that providing *set* and *get* capabilities is essentially the same as making a class's instance variables `public`. This is one

of the subtleties that makes Java desirable for software engineering.

A `public` non-`final` instance variable can be read or written by any method that has a reference to an object containing that variable. A `public` *get* method certainly allows client-code methods to access a `private` instance variable, but the *get* method can control how the client "sees" the data. For example, a *get* method might format the data it returns, shielding the client code from the actual data representation. A `public` *set* method can—and should—carefully scrutinize attempts to modify the variable's value and throw an exception if necessary. For example, attempts to *set* the day of the month to 37 or a person's weight to a negative value should be rejected. Thus, although *set* and *get* methods provide access to `private` data, the access can be carefully restricted by the method implementations, promoting good software engineering.

Classes should never have `public` non-constant data. Declaring data `public static final` enables lets you make constants available to your class's clients. For example, class `Math` offers `public static final` constants `Math.PI` and `Math.E`.

Validity Checking in Set Methods

The benefits of data integrity do not follow automatically simply because instance variables are declared `private`—you must provide validity checking. A class's *set* methods could prevent attempts to assign invalid data to objects of the class. Typically, *set* methods have the `void` return type and use exception handling to indicate attempts to assign invalid data. We discuss exception handling in detail in Chapter 10.

When appropriate, provide `public` methods to access `private` instance variables. This architecture helps hide the implementation of a class from its clients, which improves program modifiability. Using *set* and *get* methods helps you create classes that are easier to debug and maintain. If only one method performs a particular operation, such as setting an instance variable in an object, it's easier to debug and maintain the class. If the instance variable is not being set correctly, the code that actually modifies the instance variable is localized to one *set* method. Your debugging efforts can be focused on that one method.

Predicate Methods

Another common use for accessor methods is to test whether a condition is true or false—such methods are often called **predicate**

methods. An example is class `ArrayList`'s `isEmpty` method, which returns `true` if the `ArrayList` is empty and `false` otherwise. A program might test `isEmpty` before attempting to read another item from an `ArrayList`. By convention, predicate method names begin with `is` rather than `get`.

Generative AI

1 Prompt genAIs to give examples of why a Java class should never have `public` non-`final` data.
2 Prompt genAIs to list and explain applications of predicate methods in Java.

8.13 Composition

A class can have references to objects of other classes as members. This is called **composition** and is sometimes referred to as a ***has-a* relationship**. For example, an `AlarmClock` object needs to know the current time and when it's supposed to sound its alarm, so it's reasonable to include two references to `Time` objects in an `AlarmClock` object. A car *has-a* steering wheel, a brake pedal, an accelerator pedal and more.

Class `Date`

This composition example contains classes `Date` (Fig. 8.16), `Employee` (Fig. 8.17) and `EmployeeTest` (Fig. 8.18). Class `Date` declares instance variables `month`, `day` and `year` (lines 5–7) to represent a date. The constructor receives three `int` parameters. Lines 15–18 validate the `month`—if it's out-of-range, lines 16–17 throw an exception. Lines 21–25 validate the `day`. If the day is incorrect based on the number of days in the particular `month`—except February 29th which requires special leap-year testing—lines 23–24 throw an exception. Lines 28–29 perform the leap-year testing for February. If the month is February, the day is 29 and the `year` is not a leap year, lines 30–31 throw an exception. If no exceptions are thrown, then lines 34–36 initialize the `Date`'s instance variables and line 38 outputs the `this` reference as a `String`. Since `this` refers to the current `Date` object, the object's

`toString` method (lines 42–44) is called implicitly to get the object's `String` representation. In this example, we assume that the `year` value is correct—an industrial-strength `Date` class should also validate the year.

Click here to view code image

```
1  // Fig. 8.16: Date.java
2  // Date class declaration.
3
4  public class Date {
5     private int month; // 1-12
6     private int day; // 1-31 based on month
7     private int year; // any year
8
9     private static final int[] daysPerMonth =
10       {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
11
12    // constructor: confirm proper value for month and day given the year
13    public Date(int month, int day, int year) {
14       // check if month in range
15       if (month <= 0 || month > 12) {
16          throw new IllegalArgumentException(
17             "month (" + month + ") must be 1-12");
18       }
19
20       // check if day in range for month
21       if (day <= 0 ||
22          (day > daysPerMonth[month] && !(month == 2 && day == 29))) {
23          throw new IllegalArgumentException("day (" + day +
24             ") out-of-range for the specified month and year");
25       }
26
27       // check for leap year if month is 2 and day is 29
28       if (month == 2 && day == 29 && !(year % 400 == 0 ||
29             (year % 4 == 0 && year % 100 != 0))) {
30          throw new IllegalArgumentException("day (" + day +
31             ") out-of-range for the specified month and year");
32       }
33
34       this.month = month;
35       this.day = day;
36       this.year = year;
37
38       System.out.printf("Date object constructor for date %s%n", this);
39    }
40
41    // return a String of the form month/day/year
42    public String toString() {
```

```
43          return String.format("%d/%d/%d", month, day, year);
44       }
45    }
```

Fig. 8.16 | `Date` class declaration.

Class `Employee`

Class `Employee` (Fig. 8.17) has reference-type instance variables `firstName` (`String`), `lastName` (`String`), `birthDate` (`Date`) and `hireDate` (`Date`), showing that a class can have as instance variables references to objects of other classes. The `Employee` constructor (lines 11–17) takes four parameters, representing the first name, last name, birth date and hire date. The objects referenced by the parameters are assigned to an `Employee` object's instance variables. When `Employee`'s `toString` method is called, it returns a `String` containing the employee's name and the `String` representations of the two `Date` objects. Each `String` is obtained with an implicit call to the `Date` class's `toString` method.

Click here to view code image

```
1  // Fig. 8.17: Employee.java
2  // Employee class with references to other objects.
3
4  public class Employee {
5     private String firstName;
6     private String lastName;
7     private Date birthDate;
8     private Date hireDate;
9
10    // constructor to initialize name, birth date and hire date
11    public Employee(String firstName, String lastName, Date birthDate,
12       Date hireDate) {
13       this.firstName = firstName;
14       this.lastName = lastName;
15       this.birthDate = birthDate;
16       this.hireDate = hireDate;
17    }
18
19    // convert Employee to String format
20    public String toString() {
21       return String.format("%s, %s Hired: %s Birthday: %s",
22          lastName, firstName, hireDate, birthDate);
23    }
24 }
```

Fig. 8.17 | `Employee` class with references to other objects.

Class `EmployeeTest`

Class `EmployeeTest` (Fig. 8.18) creates two `Date` objects representing an `Employee`'s birthday and hire date. Line 8 creates an `Employee` and initializes its instance variables by passing to the constructor two `String`s (representing the `Employee`'s first and last names) and two `Date` objects (representing the birthday and hire date). Line 10 implicitly invokes the `Employee`'s `toString` method to display the values of its instance variables and demonstrate that the object was correctly initialized.

Click here to view code image

```
1  // Fig. 8.18: EmployeeTest.java
2  // Composition demonstration.
3
4  public class EmployeeTest {
5     public static void main(String[] args) {
6        var birth = new Date(7, 24, 1949);
7        var hire = new Date(3, 12, 1988);
8        var employee = new Employee("James", "Ramos", birth, hire);
9
10       System.out.println(employee);
11    }
12 }
```

```
Date object constructor for date 7/24/1949
Date object constructor for date 3/12/1988
Ramos, James Hired: 3/12/1988  Birthday:  7/24/1949
```

Fig. 8.18 | Composition demonstration.

Generative AI

1 Prompt genAIs to provide simple examples of Java classes containing instance variables that refer to objects of other classes.
2 Prompt genAIs with class `Date`'s code (Fig. 8.16), asking them to refactor the `Date` constructor so the leap-year test is defined in an `isLeapYear` method.

8.14 `enum` Types

In Fig. 5.4, we introduced the basic `enum` type, which defines a set of constants represented as unique identifiers. In that program, the `enum` constants represented the game's status. In this section, we discuss the relationship between `enum` types and classes. Like classes, all `enum` types are reference types. An **`enum` declaration** creates an `enum` type containing a comma-separated list of `enum` constants. The declaration may optionally include other components of traditional classes, such as constructors, fields and methods (as you'll see momentarily). Each `enum` declaration declares an `enum` class with the following restrictions:

- `enum` constants are implicitly `final` because constants cannot be modified.
- `enum` constants are implicitly `static`—since they can't be modified, only one copy of the constant is needed.

Any attempt to create an object of an `enum` type with operator `new` results in a compilation error. The `enum` constants can be used anywhere constants can be used, such as in the `case` labels of `switch` statements and to control enhanced `for` statements.

Declaring Instance Variables, a Constructor and Methods in an `enum` Type

Figure 8.19 demonstrates instance variables, a constructor and methods in an `enum` type. The `enum` declaration contains two parts —the `enum` constants and the other members of the `enum` type.

Click here to view code image

```
1  // Fig. 8.19: Book.java
2  // Declaring an enum type with a constructor and explicit instance fields
3  // and accessors for these fields
4
5  public enum Book {
6     // declare constants of enum type
7     JHTP("Java How to Program", "2025"),
8     CHTP("C How to Program", "2020"),
9     CPP20FP("C++20 for Programmers", "2023"),
10    CPPHTP("C++ How to Program", "2024"),
11    PYCDS("Intro to Python for Computer Science and Data Science", "2020"),
12    PYFP("Python for Programmers", "2020");
13
14    // instance fields
```

```
15      private final String title; // book title
16      private final String copyrightYear; // copyright year
17
18      // enum constructor
19      Book(String title, String copyrightYear) {
20         this.title = title;
21         this.copyrightYear = copyrightYear;
22      }
23
24      // accessor for field title
25      public String getTitle() {
26         return title;
27      }
28
29      // accessor for field copyrightYear
30      public String getCopyrightYear() {
31         return copyrightYear;
32      }
33   }
```

Fig. 8.19 | Declaring an `enum` type with a constructor and explicit instance fields and accessors for these fields.

Lines 7–12 declare six constants. Each is optionally followed by arguments that are passed to the **`enum` `Book`'s constructor** (lines 19–22). Like the constructors in classes, an `enum` constructor can specify any number of parameters and can be overloaded. In this example, the `enum` constructor requires two `String` parameters, specifying the book's title and copyright year. To properly initialize each `enum` constant, we follow it with parentheses containing two `String` arguments. In an `enum` declaration, the `enum` constants must be declared before the `enum`'s constructors, fields and methods.

The second part (lines 15–32) declares the `enum`'s other members—instance variables `title` and `copyrightYear` (lines 15–16), a constructor (lines 19–22) and two methods (lines 25–27 and 30–32) that return the book title and copyright year, respectively. Each constant in type `Book` is an object of type `Book` that has its own copy of instance variables.

Using `enum` type `Book`

Figure 8.20 tests the `Book` `enum` and illustrates how to iterate through a range of its constants. For every `enum`, the compiler generates the **`static` method `values`** (called in line 10), which returns an array of the `enum`'s constants in their declaration order. Lines 10–13 display the constants. Line 12 invokes the `enum` `Book`'s `getTitle` and `getCopyrightYear` methods to get the title and

copyright year associated with the constant. When an `enum` constant is implicitly converted to a `String` (e.g., `book` in line 11), the constant's identifier is used as the `String` representation (e.g., `JHTP` for the first `enum` constant).

Click here to view code image

```
1  // Fig. 8.20: EnumTest.java
2  // Testing enum type Book.
3  import java.util.EnumSet;
4
5  public class EnumTest {
6     public static void main(String[] args) {
7        System.out.println("All books:");
8
9        // print all books in enum Book
10       for (Book book : Book.values()) {
11          System.out.printf("%-10s%-57s%s%n", book,
12             book.getTitle(), book.getCopyrightYear());
13       }
14
15       System.out.printf("%nDisplay a range of enum constants:%n");
16
17       // print first four books
18       for (Book book : EnumSet.range(Book.JHTP, Book.CPPHTP)) {
19          System.out.printf("%-10s%-57s%s%n", book,
20             book.getTitle(), book.getCopyrightYear());
21       }
22    }
23 }
```

```
All books:
JHTP      Java How to Program                                       2025
CHTP      C How to Program                                          2020
CPP20FP   C++20 for Programmers                                     2023
CPPHTP    C++ How to Program                                        2024
PYCDS     Intro to Python for Computer Science and Data Science     2020
PYFP      Python for Programmers                                    2020

Display a range of enum constants:
JHTP      Java How to Program                                       2025
CHTP      C How to Program                                          2020
CPP20FP   C++20 for Programmers                                     2023
CPPHTP    C++ How to Program                                        2024
```

Fig. 8.20 | Testing `enum` type `Book`.

Lines 18–21 use the **`static` method `range` of class `EnumSet`** (package `java.util`) to display a range of the `enum` `Book`'s

constants. Method `range` takes two parameters—the first and the last `enum` constants in the range—and returns an `EnumSet` that contains all the constants between these two constants, inclusive. For example, the expression `EnumSet.range(Book.JHTP, Book.CPPHTP)` returns an `EnumSet` containing `Book.JHTP`, `Book.CHTP`, `Book.CPP20FP` and `Book.CPPHTP`. The enhanced `for` statement can be used with an `EnumSet` just as it can with an array. Lines 18–21 use it to display the title and copyright year of every book in the `EnumSet`. Class `EnumSet` provides several other `static` methods for creating sets of `enum` constants from the same `enum` type.

Generative AI

1 Prompt genAIs with the code of Figs. 8.19–8.20, asking them to explain and provide examples of the `EnumSet` methods `of` and `complementOf`.

8.15 Garbage Collection

Every object uses system resources, such as memory. We need a disciplined way to give resources back to the system when they're no longer needed; otherwise, "resource leaks" might occur that would prevent resources from being reused by your program or others. The JVM performs automatic **garbage collection** to reclaim the memory occupied by objects that are no longer used.

If there are no more references to an object, the object is eligible to be collected when the JVM executes its **garbage collector**. This might not occur right away. If a program terminates before garbage collection occurs, all of its memory is returned to the system. So, memory leaks common in other languages like C and C++ (because they do not automatically reclaim memory) are less likely in Java. However, some can still happen in subtle ways.

Resource leaks other than memory leaks can also occur. For example, a program may open a file on secondary storage to modify its contents. If the program does not close the file, it must terminate before another program can use the file.

Many Java API classes (e.g., class `Scanner` and classes that read from or write to files) provide `close` or `dispose` methods that can release resources when they're no longer needed in a program.

Generative AI

1 Prompt genAIs for an overview of the garbage collectors that come with the latest version of the OpenJDK. Ask which is the default and why developers might choose another. Several organizations provide JVMs. Each typically has multiple garbage collectors.

8.16 `static` Class Members

Every object has its own copy of its class's instance variables. Sometimes, one variable should be shared by all objects of a class. A **`static` field**—called a class variable—represents **classwide information**. All objects of the class share the class's `static` fields.

Motivating `static`

Let's motivate `static` data with an example. Suppose we have a video game with `Martian`s and other space creatures. Each `Martian` tends to be brave and willing to attack other space creatures when the `Martian` is aware that at least four other `Martian`s are present. If fewer than five `Martian`s are present, each becomes cowardly. Thus, each `Martian` needs to know the `martianCount`. We could endow class `Martian` with `martianCount` as an instance variable. Then, every `Martian` would have a separate copy of the instance variable, and every time we create a new `Martian` or destroy an existing one, we'd have to update the `martianCount` in every `Martian` object. This wastes space with the redundant copies, wastes time updating the separate copies and is error-prone. Instead, we declare `martianCount` to be `static`, making `martianCount` classwide data so every `Martian` can access it. This saves space. The `Martian` constructor would increment the `static martianCount` as each new `Martian` is created. Since there's only one copy, we do not have to increment separate copies for each `Martian` object. Similarly, when a `Martian` is destroyed, a method of the class would be called to decrement the `martianCount`.

Class Scope

A class's `static` variables have class scope—they can be used in all

of the class's methods. We can access a class's `public static` members through a reference to any object of the class or by qualifying the member name with the class name and a dot (`.`), as in `Math.sqrt(2)`. A class's `private static` class members can be accessed by client code only through the class's methods. The `static` class members exist even when no objects of the class exist—they're available as soon as the class is loaded into memory at execution time. To access a `public static` member when no objects of the class exist (and when they do), prefix the class name and a dot (`.`) to the `static` member, as in `Math.PI`. To access a `private static` member when no objects of the class exist, provide a `public static` method and call it by qualifying its name with the class name and a dot.

`static` Methods Cannot Directly Access Instance Variables and Instance Methods

A `static` method cannot directly access a class's instance variables and instance methods because a `static` method can be called even when no objects of the class have been instantiated. So, "`this`." cannot be used in a `static` method. Java creates the `this` reference only in instance methods because it must refer to a specific object of the class. When a `static` method is called, there might not be any objects of its class in memory.

Tracking the Number of Employee Objects That Have Been Created

Our next program declares two classes—`Employee` (Fig. 8.21) and `EmployeeTest` (Fig. 8.22). Class `Employee` declares `private static` variable `count` (Fig. 8.21, line 6) and `public static` method `getCount` (lines 32–34). The `static` variable `count` maintains a count of the number of `Employee` objects created so far. This class variable is initialized to `0` in line 6. If a `static` variable is not initialized, the compiler assigns it a default value—in this case, `0`, the default value for type `int`.

Click here to view code image

```
1  // Fig. 8.21: Employee.java
2  // static variable used to maintain a count of the number of
3  // Employee objects in memory.
4
5  public class Employee {
6     private static int count = 0; // number of Employees created
```

```
7       private String firstName;
8       private String lastName;
9
10      // initialize Employee, add 1 to static count and output
11      // a String indicating that the constructor was called
12      public Employee(String firstName, String lastName) {
13         this.firstName = firstName;
14         this.lastName = lastName;
15
16         ++count; // increment static count of employees
17         System.out.printf("Employee constructor: %s %s; count = %d%n",
18            firstName, lastName, count);
19      }
20
21      // get first name
22      public String getFirstName() {
23         return firstName;
24      }
25
26      // get last name
27      public String getLastName() {
28         return lastName;
29      }
30
31      // static method to get static count value
32      public static int getCount() {
33         return count;
34      }
35   }
```

Fig. 8.21 | `static` variable used to maintain a count of the number of `Employee` objects in memory.

When `Employee` objects exist, the variable `count` can be used in any method of an `Employee` object. This example increments `count` in the constructor (line 16). The `public static` method `getCount` (lines 32–34) returns the number of `Employee` objects created so far. When no objects of class `Employee` exist, client code can access variable `count` by calling method `getCount` via the class name, as in `Employee.getCount()`. Invoke every `static` method using the class name and a dot (`.`) to emphasize that the method is `static`.

Class `EmployeeTest`

`EmployeeTest` method `main` (Fig. 8.22) instantiates two `Employee` objects (lines 11–12). When each `Employee` object's constructor is invoked, it initializes the `Employee`'s instance variables `firstName` and `lastName`, then adds one to the `Employee` `static` variable

count.

Click here to view code image

```
1  // Fig. 8.22: EmployeeTest.java
2  // static member demonstration.
3
4  public class EmployeeTest {
5     public static void main(String[] args) {
6        // show that count is 0 before creating Employees
7        System.out.printf("Employees before instantiation: %d%n",
8           Employee.getCount());
9
10       // create two Employees; count should then be 2
11       var e1 = new Employee("Anna", "Gruber");
12       var e2 = new Employee("Teva", "Maihi");
13
14       // show that count is 2 after creating two Employees
15       System.out.printf("%nEmployees after instantiation:%n");
16       System.out.printf("via e1.getCount(): %d%n", e1.getCount());
17       System.out.printf("via e2.getCount(): %d%n", e2.getCount());
18       System.out.printf("via Employee.getCount(): %d%n",
19          Employee.getCount());
20
21       // get names of Employees
22       System.out.printf("%nEmployee 1: %s %s%nEmployee 2: %s %s%n",
23          e1.getFirstName(), e1.getLastName(),
24          e2.getFirstName(), e2.getLastName());
25    }
26 }
```

```
Employees before instantiation: 0
Employee constructor: Anna Gruber; count = 1
Employee constructor: Teva Maihi; count = 2

Employees after instantiation:
via e1.getCount(): 2
via e2.getCount(): 2
via Employee.getCount(): 2

Employee 1: Anna Gruber
Employee 2: Teva Maihi
```

Fig. 8.22 | `static` member demonstration.

When `main` terminates, local variables `e1` and `e2` are discarded—remember that a local variable exists only until the block in which it's declared completes execution. Because `e1` and `e2` were the only references to the `Employee` objects created in lines 11–12

(Fig. 8.22), these objects become "eligible for garbage collection" as `main` terminates.

In a typical app, the garbage collector might eventually reclaim the memory for any objects that are eligible for collection. If any objects are not reclaimed before the program terminates, the operating system will reclaim the memory used by the program. The JVM does not guarantee when, or even if, the garbage collector will execute. When it does, it's possible that no objects or a subset of the eligible objects will be collected.

Generative AI

1 Prompt genAIs for some simple use-cases of `static` class variables in Java.

8.17 `static` Import

In Section 5.8, you learned about the `static` fields and methods of class `Math`—we access these by preceding each with the class name `Math` and a dot (`.`). A **`static` import declaration** enables you to import a class's `static` members so you can access them via their unqualified names in your class—without the class name and a dot (`.`).

`static` Import Forms

A `static` import declaration has two forms that are listed in the Java Language Specification's language grammar as ***SingleStaticImportDeclaration*** and ***StaticImportOnDemand-Declaration***:

- A *SingleStaticImportDeclaration* imports a specific `static` member.
- A *StaticImportOnDemandDeclaration* imports all `static` members of a class.

A *SingleStaticImportDeclaration* consists of a package name (e.g., `java.lang`), a class name and the name of a `static` method or field, as in:

Click here to view code image

```
import static packageName.ClassName.staticMemberName;
```

A *StaticImportOnDemandDeclaration* consists of a package name (e.g., `java.lang`), a class name and an asterisk (`*`) indicating that *all* `static` members of the class should be available for use in the file, as in:

Click here to view code image

```
import static packageName.ClassName.*;
```

Such declarations import only `static` class members. Regular `import` statements should be used to specify the classes used in a program.

Demonstrating `static` Import

Figure 8.23 demonstrates a `static` import. Line 3 is a *StaticImportOnDemandDeclaration*, which imports all of class `Math`'s `static` fields and methods. Lines 7–10 access the `Math` class's `static` methods `sqrt` (line 7) and `ceil` (line 8) and its `static` fields `E` (line 9) and `PI` (line 10) without qualifying their names with class name `Math` and a dot.

Click here to view code image

```
1  // Fig. 8.23: StaticImportTest.java
2  // Static import of Math class methods.
3  import static java.lang.Math.*;
4
5  public class StaticImportTest {
6     public static void main(String[] args) {
7        System.out.printf("sqrt(900.0) = %.1f%n", sqrt(900.0));
8        System.out.printf("ceil(-9.8) = %.1f%n", ceil(-9.8));
9        System.out.printf("E = %f%n", E);
10       System.out.printf("PI = %f%n", PI);
11    }
12 }
```

```
sqrt(900.0) = 30.0
ceil(-9.8) = -9.0
E = 2.718282
PI = 3.141593
```

Fig. 8.23 | `static` import of `Math` class methods.

Generative AI

1 Prompt genAIs, asking for insights on when to use Java's *StaticImportOnDemandDeclarations* and *SingleStaticImportDeclarations*.

8.18 `final` Instance Variables

The principle of least privilege is fundamental to good software engineering. In the context of an app's code, it states that code should be granted only the amount of privilege and access needed to accomplish its designated operation, but no more. This makes your programs more robust by preventing code from accidentally (or maliciously) modifying variable values and calling methods that should not be accessible.

Let's see how this principle applies to instance variables. Some should be modifiable, and some should not. You can use the keyword `final` to specify that a variable is not modifiable (i.e., it's a constant) and that any attempt to modify it is an error. For example,

```
private final int INCREMENT;
```

declares a `final` (constant) `int` instance variable `INCREMENT`. Such variables can be initialized when they're declared. If they're not, they must be initialized in every constructor and cannot be changed after the object is constructed. Initializing constants in constructors enables each object of the class to have a different value for the constant. A compilation error occurs if a `final` variable is not initialized in its declaration or in every constructor. By convention, identifiers for constants use all capital letters with words separated by underscores (_).

Attempting to modify a `final` instance variable after it's initialized is a compilation error caught at compilation time rather than execution time. It's always preferable to get bugs out at compilation time, if possible, rather than at execution time (where experience has shown that repair is often far more costly).

Declaring an instance variable as `final` helps enforce the principle of least privilege. If an instance variable should not be modified, declare it `final` to prevent modification. For example, in Fig. 8.17, the instance variables `firstName`, `lastName`, `birthDate`

and `hireDate` are never modified after initialization, so they should be declared `final`. Testing, debugging and maintaining programs is easier when every variable that can be `final` is, in fact, `final`. You'll see additional benefits of `final` in Chapter 18, Concurrency: Platform Threads to Virtual Threads.

A `final` field should be `static` if it's initialized in its declaration to a value that's the same for all objects of the class. Making the field `static` enables all objects of the class to share the `final` field.

Generative AI

1 Prompt genAIs for examples that demonstrate the principle of least privilege in Java programming.
2 Prompt genAIs to explain the naming conventions for Java constants.

8.19 Package Access

If a method or field of a class is not declared with an access modifier, it has **package access** by default. In a program with one class, this has no effect. In a program with multiple classes in the same package, the classes can access each other's package-access members directly through references to objects of those classes or through the class name for `static` members. Package access is rarely used.

Figure 8.24 demonstrates package access. The program contains two classes in one source-code file—class `PackageDataTest` (lines 5–19) and class `PackageData` (lines 22–30). Classes in the same source file are part of the same package. So, `PackageDataTest` is allowed to modify the package-access data of `PackageData` objects. In the `PackageData` class declaration, lines 23–24 declare the instance variables `number` and `string` with no access modifiers—so, these are package-access instance variables. Class `PackageDataTest`'s `main` method creates a `PackageData` object (line 7) then modifies the `PackageData` instance variables directly (lines 13–14). When you compile this program, the compiler produces the files `PackageDataTest.class` and `PackageData.class` and places them in the same folder.

Click here to view code image

```
1 // Fig. 8.24: PackageDataTest.java
```

```
// Package-access members of a class are accessible by other classes
// in the same package.

public class PackageDataTest {
   public static void main(String[] args) {
      var packageData = new PackageData();

      // output String representation of packageData
      System.out.printf("After instantiation:%n%s%n", packageData);

      // change package access data in packageData object
      packageData.number = 77;
      packageData.string = "Goodbye";

      // output String representation of packageData
      System.out.printf("%nAfter changing values:%n%s%n", packageData);
   }
}

// class with package access instance variables
class PackageData {
   int number = 0; // package-access instance variable
   String string = "Hello"; // package-access instance variable

   // return PackageData object String representation
   public String toString() {
      return String.format("number: %d; string: %s", number, string);
   }
}
```

```
After instantiation:
number: 0; string: Hello

After changing values:
number: 77; string: Goodbye
```

Fig. 8.24 | Package-access members of a class are accessible by other classes in the same package.

Generative AI

1 Prompt genAIs, asking why package access is rarely used in Java.

8.20 `record` Classes

Some programming languages provide a fixed-size, immutable data structure called a tuple that simply groups related data items. In those languages, you can pass tuple objects to methods, return them from methods and conveniently access their data items. They basically carry data from one portion of a program to another. Java's **`record` classes**[8] provide a compact syntax for declaring **data-carrier classes** that enable you to create objects and use them similarly to tuples.

8. "JEP 395: Records." Accessed July 12, 2024. `https://openjdk.org/jeps/395`.

When declaring a `record` class, you specify its instance variable names and types, and the Java compiler conveniently autogenerates "boilerplate" code for you, including:

- a `private final` instance variable for each of the related data items you specify—a `record`'s instance variables are immutable,[9]

  9. In Chapter 18, Concurrency: Platform Threads to Virtual Threads, we'll discuss various benefits of immutable objects in programs that perform operations concurrently.

- a `public` constructor with arguments to initialize each instance variable—this constructor does not validate its arguments, but you can provide validation code,
- *get* methods with the same names as the instance variables—each returns its corresponding instance variable's value,
- a `toString` method that returns a `record`'s `String` representation in the following form, where *variable1*, *variable2*, … are the instance variable names:

  Click here to view code image

  TypeName[*variable1* = *value1*, *variable2* = *value2*, …]

- an `equals` method that returns `true` or `false`, based on whether the contents of two `record` objects of the same type have the same contents, and
- a `hashCode` method—we discuss this method and the data structures concept known as "hashing" in Section 12.10.

Defining a `record` Class

Figure 8.25 defines the `record` class `Employee` (lines 3–15), representing an employee's first name (a `String`), last name (a `String`), salary (a `double` for simplicity in this example) and department (a `String`). We broke the program into pieces for discussion purposes. A `record` class begins with the **`record` keyword** and a type name followed by parentheses containing the class's instance variables. After the parentheses, the `record` class's body is delimited by braces (`{` and `}`). The body's braces will be empty for `record` classes that require only the default compiler-generated code.

Click here to view code image

```
1  // Fig. 8.25: Records.java
2  // Demonstrating an Employee record class.
3  record Employee(
4     String first,
5     String last,
6     double salary,
7     String department) {
8
9     // compact constructor for validation
10    Employee {
11       if (salary < 0.0) {
12          throw new IllegalArgumentException("salary must be >= 0.0");
13       }
14    }
15 }
16
```

Fig. 8.25 | Demonstrating an `Employee` `record` class.

You can customize the compiler-generated constructor and methods. For the constructor, `record`s use a shorthand notation (lines 10–14) consisting of the `record` class's name followed immediately by braces containing the constructor's data validation code—this code is added to the autogenerated constructor. If the data validation does not throw an exception, the constructor automatically assigns each constructor argument to the corresponding instance variable.

Creating `Employee` `record` Objects and Getting Their `String` Representations

Lines 19–21 create three new `Employee` objects. As with objects of regular classes, these statements use the `record` class's constructor

to initialize each object's data. Line 24 displays the autogenerated `String` representation of each `Employee` object—converting any object to a `String` (`%s`) in a `printf` call implicitly calls the object's `toString` method.

Click here to view code image

```
17  public class Records {
18     public static void main(String[] args) {
19        Employee e1 = new Employee("Prince", "Botha", 5000, "IT");
20        Employee e2 = new Employee("Myra", "Singh", 7600, "SALES");
21        Employee e3 = new Employee("Myra", "Singh", 7600, "SALES");
22
23        System.out.println("Getting Employee string representations:");
24        System.out.printf("e1: %s%ne2: %s%ne3: %s%n%n", e1, e2, e3);
25
```

```
Getting Employee string representations:
e1: Employee[first=Prince, last=Botha, salary=5000.0, department=IT]
e2: Employee[first=Myra, last=Singh, salary=7600.0, department=SALES]
e3: Employee[first=Myra, last=Singh, salary=7600.0, department=SALES]
```

Getting an `Employee` Object's Instance Variable Values

Lines 27–30 demonstrate `record` class `Employee`'s autogenerated *get* methods for the object `e1`—again, each has the same name as its corresponding instance variable.

Click here to view code image

```
26        System.out.println("Getting data from e1:");
27        System.out.printf("first: %s%n", e1.first());
28        System.out.printf("last: %s%n", e1.last());
29        System.out.printf("salary: %.2f%n", e1.salary());
30        System.out.printf("department: %s%n%n", e1.department());
31
```

```
Getting data from e1:
first: Prince
last: Botha
salary: 5000.00
department: IT
```

Comparing `Employee` Objects for Equality

Lines 33–34 demonstrate `record` class `Employee`'s autogenerated `equals` method. This method compares the values of the instance variables in the object that calls `equals` to the corresponding instance variables in its argument and returns `true` if they are equal; otherwise, it returns `false`. In this example, `e1` and `e2` are not equal, and `e2` and `e3` are equal.

Click here to view code image

```
32          System.out.println("Comparing Employee objects for equality:");
33          System.out.printf("e1.equals(e2): %b%n", e1.equals(e2));
34          System.out.printf("e2.equals(e3): %b%n%n", e2.equals(e3));
35
```

```
Comparing Employee objects for equality:
e1.equals(e2): false
e2.equals(e3): true
```

`Employee` Constructor Throwing an Exception for an Invalid Salary

Line 37 attempts to create an `Employee` with an invalid salary. In this case, the `Employee` constructor's custom validation code (lines 11–13) recognizes the invalid data and throws an exception, showing that the error occurred in line 12, which was called from line 37:

Click here to view code image

```
36          System.out.println("Creating Employee with invalid salary:");
37          Employee e4 = new Employee("Isabella", "Garcia", -1000, "IT");
38      }
39  }
```

```
Creating Employee with invalid salary:
Exception in thread "main" java.lang.IllegalArgumentException: salary must be
>= 0.0
        at Employee.<init>(Records.java:12)
        at Records.main(Records.java:37)
```

`record` Notes

Java `record`s can also declare `static` data, `static` methods and instance methods, and can implement interfaces (Chapter 9). Unlike regular Java classes, `record` classes may be declared in a method's

body for use only in that method. For example, a method that reads and processes data from a file might use a local `record` class to create objects for storing and processing the file's records.

Generative AI

1 Declare a `record` class called `Part` representing a hardware store inventory item containing a `name` (`String`), `price` (`BigDecimal`) and a `quantity` (`int`). Prompt genAIs to modify your code to include a shorthand constructor that

a. validates the price, ensuring it's greater than 0.0,

b. validates the quantity, ensuring it's non-negative and

c. throws an `IllegalArgumentException` if either is invalid.

Also prompt genAIs to write a program that tests the updated `record` class, including invalid values when initializing `Part` objects.

8.20.1 Pattern Matching for `switch` Expressions

Since `switch` expressions were introduced in Java 14, they've been enhanced with reference-type pattern-matching capabilities in Java 21.[10] Each `case` can specify a reference type and a variable name to which the controlling-expression object is assigned if its type matches the type in a given `case`. That `case` can then manipulate the object via the variable. Figure 8.26 defines `record` classes for a `Rectangle` (line 6) with instance variables `length` and `width`, and a `Circle` (line 7) with instance variable `radius`. We use these with the `getPerimeter` method (lines 20–26) to calculate a `Rectangle`'s and a `Circle`'s perimeters. Lines 10–11 in `main` create and initialize a `Rectangle` and a `Circle`. Lines 12–15 display each object's `String` representation and perimeter.

Click here to view code image

```
1  // Fig. 8.26: RecordPatternMatchingDemo.java
2  // Assigning result of a switch expression to a variable.
3
4  public class RecordPatternMatchingDemo {
5     // record classes representing data for a Rectangle and a Circle
6     record Rectangle(double length, double width) { }
```

```
7       record Circle(double radius) { }
8
9       public static void main(String[] args) {
10         Rectangle r = new Rectangle(10, 5);
11         Circle c = new Circle(10);
12         System.out.printf("Rectangle r: %s%nperimeter: %.2f%n%n",
13            r, getPerimeter(r));
14         System.out.printf("Circle c : %s%nperimeter: %.2f%n",
15            c, getPerimeter(c));
16      }
17
18      // uses pattern-matching switch to calculate perimeter of a
19      // Rectangle or Circle; throws IllegalArgumentException for other types
20      public static double getPerimeter(Object shape) {
21         return switch (shape) {
22            case Rectangle r -> 2 * r.length() + 2 * r.width();
23            case Circle c -> 2 * Math.PI * c.radius() ;
24            default -> throw new IllegalArgumentException("invalid type");
25         };
26      }
27   }
```

```
Rectangle r: Rectangle[length=10.0, width=5.0]
perimeter: 30.00

Circle c : Circle[radius=10.0]
perimeter: 62.83
```

Fig. 8.26 | Assigning result of a `switch` expression to a variable.

10. "JEP 441: Pattern Matching for `switch`." Accessed July 18, 2024. https://openjdk.org/jeps/441.

Method `getPerimeter` (lines 20–26) receives an `Object` and returns the calculation result as a `double`. The parameter type `Object` indicates that the method can receive any reference-type object as an argument. The **pattern-matching `switch`** (lines 21–25) determines the type of the object in its controlling expression:

- If its type matches a `case`, the `switch` stores the object in that `case`'s variable.
- Then, the expression to the right of the matching `case`'s `->` uses the variable to get the `record` object's data and perform the appropriate perimeter calculation.

The `default` case executes for any type the `switch` does not recognize. This example `throw`s an `IllegalArgumentException` for anything that is not a `Rectangle` or a `Circle`.

8.20.2 Decomposing `records` in `switch` Expressions

In pattern-matching `switch` expressions, you can use a **`record` pattern**[11] to automatically assign the values in a `record` object to variables used in the `case`. This is known as **decomposing `records`**, and it enables you to access the data items without having to call the `record`'s *get* methods. Figure 8.27 modifies Fig. 8.26, using `record` patterns (lines 23–24) to decompose the `Rectangle` and `Circle` objects. You define a `record` pattern as the `record` class name followed by parentheses containing the variables in which to place a matching object's values. The pattern's variable names need not match the `record`'s instance variable names. The `case` can then process the matching object's data using the pattern's variables. We used local-variable type inference (`var`) here to infer the pattern's variable types from the `record` classes' instance variables. Figure 8.27 produces the same results as Fig. 8.26.

11. "JEP 440: Record Patterns." Accessed March 24, 2024. `https://openjdk.org/jeps/440`.

Click here to view code image

```
1  // Fig. 8.27: RecordPatternMatchingDemo2.java
2  // Using record patterns in a switch expression to assign
3  // record members to variables for use in a given case.
4
5  public class RecordPatternMatchingDemo2 {
6     // record classes representing data for a Rectangle and a Circle
7     record Rectangle(double length, double width) { }
8     record Circle(double radius) { }
9
10    public static void main(String[] args) {
11       Rectangle r = new Rectangle(10, 5);
12       Circle c = new Circle(10);
13       System.out.printf("Rectangle r: %s%nperimeter: %.2f%n%n",
14          r, getPerimeter(r));
15       System.out.printf("Circle c : %s%nperimeter: %.2f%n",
16          c, getPerimeter(c));
17    }
18
19    // uses pattern-matching switch to calculate perimeter of a
20    // Rectangle or Circle; throws IllegalArgumentException for other types
21    public static double getPerimeter(Object shape) {
22       return switch (shape) {
23          case Rectangle(var length, var width) -> 2 * length + 2 * width;
24          case Circle(var radius) -> 2 * Math.PI * radius;
```

```
            default -> throw new IllegalArgumentException("invalid type");
        };
    }
}
```

```
Rectangle r: Rectangle[length=10.0, width=5.0]
perimeter: 30.00

Circle c : Circle[radius=10.0]
perimeter: 62.83
```

Fig. 8.27 | Using `record` patterns in a `switch` expression to assign `record` members to variables for use in a given case.

8.21 Wrap-Up

In this chapter, you declared custom classes, created objects of those classes and called their methods to perform useful actions. You declared instance variables to maintain data for each object and methods to operate on that data. We discussed the difference between a method's local variables and a class's instance variables, and you saw that instance variables are initialized automatically to default values. You used constructors to specify initial values for an object's instance variables.

We presented a card-shuffling-and-dealing simulation. The `Time` class case study showed a complete class declaration consisting of `private` data, overloaded `public` constructors for initialization flexibility, *set* and *get* methods for manipulating the class's data, and methods that returned `String` representations of a `Time` object in standard time and universal time formats. You saw that a class's `toString` method returns an object's `String` representation and can be called explicitly or implicitly when an object is used where a `String` is expected. We showed how to `throw` an exception to indicate that a problem has occurred.

You learned that the `this` reference is used implicitly in a class's instance methods to access the class's instance variables and other instance methods. You also saw explicit uses of `this` to access the class's members (including shadowed fields) and, from a constructor, to call another constructor of the same class.

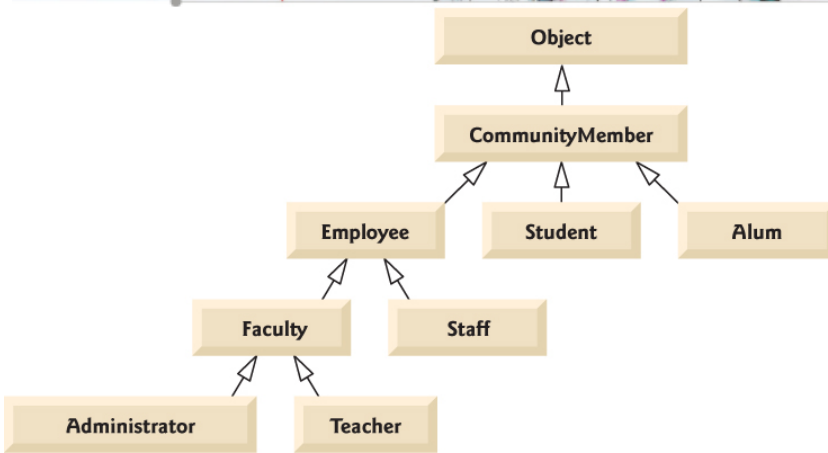
We discussed the differences between default constructors provided by the compiler and no-argument constructors provided by the programmer. You saw that with composition, a class can

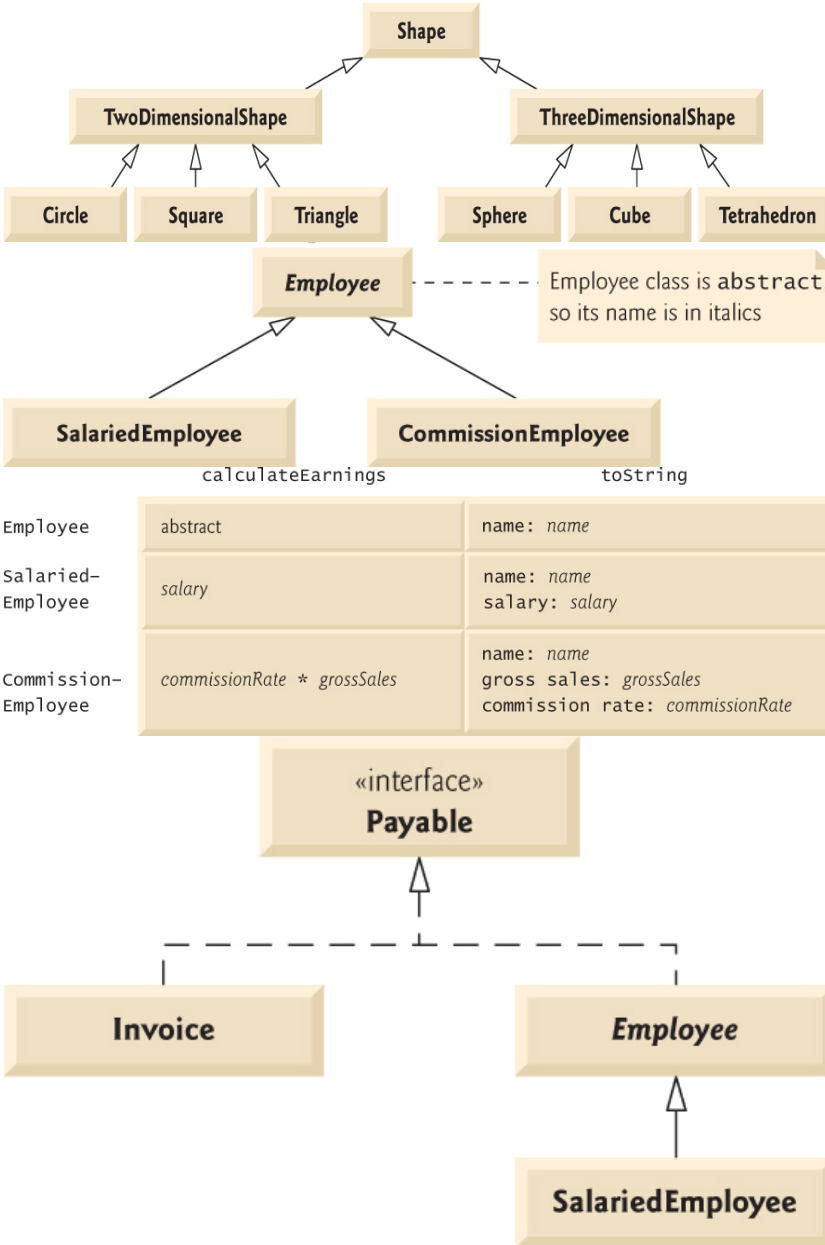
have references to objects of other classes as members. We discussed more features of `enum` types. We mentioned Java's garbage-collection capability and how it (unpredictably) reclaims the memory of objects that are no longer used. We motivated `static` fields, and demonstrated how to declare and use `static` fields and methods. You also saw how to declare and initialize `final` fields.

We discussed package access for fields and methods without an access modifier. You saw the relationship between classes in the same package that allows each class to access the package-access members of other classes in the package.

We introduced `record` classes and showed that the Java compiler autogenerates various common methods for them. We also showed how `record` objects can be used with `switch` expressions to perform different operations based on `record` class types. Finally, you used `record` patterns to decompose `record` objects into their individual data items.

We've now introduced the basic concepts of control statements, methods, arrays, collections, classes and objects. The next chapter discusses a core aspect of object-oriented programming in Java—inheritance. You'll see that all classes in Java are related by inheritance, directly or indirectly, to the class called `Object`. You'll explore how the relationships among classes enable you to build powerful applications.


Object
CommunityMember
Employee
Student
Alum
Faculty
Staff
Administrator
Teacher


Shape
TwoDimensionalShape
ThreeDimensionalShape
Circle
Square
Triangle
Sphere
Cube
Tetrahedron
Employee
Employee class is abstract, so its name is in italics
SalariedEmployee
CommissionEmployee
calculateEarnings
toString
Employee
abstract
name: name
Salaried-Employee
salary
name: name
salary: salary
Commission-Employee
commissionRate * grossSales
name: name
gross sales: grossSales
commission rate: commissionRate
«interface»
Payable
Invoice
Employee
SalariedEmployee

Real-World Modeling with Inheritance, Polymorphism & Interfaces

Objectives

In this chapter, you'll:

- Use inheritance to develop new classes based on existing ones.
- Understand the relationship between superclasses and subclasses.
- Create a class that `extends` another class to inherit its attributes and behaviors.
- Call a superclass method from a subclass override via `super`.
- Learn how constructors are used in inheritance hierarchies.
- Learn about class `Object`, the direct or indirect superclass of all classes.
- Observe how polymorphism can make programming more convenient and systems more easily extensible.
- Use overridden methods to effect polymorphism.
- Distinguish between concrete and `abstract` classes.
- Declare `abstract` methods and `abstract` classes.
- Declare and implement interfaces and become familiar with `default`, `static` and `private` interface methods.
- Learn how `protected` gives subclass methods direct access to

superclass members.

- Use `sealed` to control which classes can extend a class or implement an interface.

Outline

9.1 Introduction

9.2 Superclasses and Subclasses

9.3 Relationship Between Superclasses and Subclasses

9.3.1 Creating and Using a `SalariedEmployee` Class

9.3.2 Creating a `SalariedEmployee`/`SalariedCommissionEmployee` Inheritance Hierarchy

9.4 Class `Object`

9.5 Intro to Polymorphism: Polymorphic Video Game

9.6 Demonstrating Polymorphic Behavior

9.7 `abstract` Classes and Methods

9.8 Case Study: Payroll System Using Polymorphism

9.8.1 `abstract` Superclass `Employee`

9.8.2 Concrete Subclass `SalariedEmployee`

9.8.3 Concrete Subclass `CommissionEmployee`

9.8.4 Polymorphic Processing, Operator `instanceof` and Downcasting

9.9 `final` Methods and Classes

9.10 Issues with Constructors Calling Instance Methods

9.11 Creating and Using Interfaces

9.11.1 Developing a `Payable` Hierarchy

9.11.2 Interface `Payable`

9.11.3 Class `Invoice`

9.11.4 Modifying Class `Employee` to Implement Interface `Payable`

9.11.5 Using Interface `Payable` to Process `Invoices` and `Employees` Polymorphically

9.11.6 Some Common Java API Interfaces

9.12 Other Interface Features

9.12.1 `default` Interface Methods

9.12.2 `static` Interface Methods
9.12.3 Functional Interfaces
9.12.4 `private` Interface Methods
9.13 Program to an Interface, Not an Implementation
9.13.1 `CompensationModel` Interface
9.13.2 An `Employee` Has a `CompensationModel`—Composition and Dependency Injection
9.13.3 `CompensationModel` Implementations
9.13.4 Testing the `CompensationModel` Hierarchy
9.13.5 Dependency Injection Design Benefits
9.13.6 `interface`s vs. `abstract` Classes
9.14 `sealed` Classes and Interfaces
9.15 `private` Constructors
9.16 `protected` Members
9.17 Wrap-Up
Special Section: Generative AI Exercises

9.1 Introduction

In Chapter 1, we introduced the basics of object-oriented programming (OOP) in the context of real-world things—automobiles. With the objects-natural approach, you've been creating objects from preexisting classes and conveniently getting them to perform powerful operations without having to know how they're built on the inside. This is much like driving an automobile without knowing how to build one.

In Chapter 8, you started building custom classes, which are software models of real-world things. Here in Chapter 9, you'll study the key object-oriented technologies of inheritance, polymorphism and interfaces, which enable you to model the relationships and interactions among real-world objects.

Inheritance

With inheritance, you can base new classes on existing proven

software—a new class is created by acquiring an existing class's capabilities and possibly modifying them or adding new ones. As you'll see, this is done without copying and pasting source code.

When creating a class, rather than declaring completely new members, you can specify that the new class should inherit the members of an existing one. The existing class is the **superclass**, and the new class is the **subclass.** A subclass can add its own fields and methods, making it more specialized than its superclass. The subclass exhibits the behaviors of its superclass. It can override them to operate appropriately for the subclass, so inheritance is sometimes called **specialization**. A subclass can become a superclass for future subclasses.

We distinguish between the inheritance ***is-a* relationship** and composition's *has*-a relationship. In an *is-a* relationship, an object of a subclass can also be treated as an object of its superclass. For example, a car *is a* vehicle, and a square *is a* shape. In a *has-a* relation-ship, an object contains as members references to other objects. For example, a car *has a* steering wheel (and a car object *has a* reference to a steering-wheel object).

New classes can inherit from abundant classes in class libraries. Organizations develop their own class libraries and can use others (mostly open-source) available worldwide. This facilitates the rapid development of more powerful, proven and economical software.

Polymorphism

We demonstrate **polymorphism** with inheritance hierarchies, which enables you to conveniently "program in the general" rather than "program in the specific." You can write programs with objects that share the same superclass, either directly or indirectly, as if they were all objects of that superclass—this can simplify programming. Polymorphism's key concept is relying on each object to "do the right thing"—i.e., what's appropriate for that type of object in response to the same method call. Thus, the same message sent to various types of objects can have many forms of results, hence the term polymorphism.

With polymorphism, we can design and implement systems that are easily extensible—new classes can be added with little or no modification to the general portions of the program. The new classes simply "plug right in."

Occasionally, when performing polymorphic processing, we might want to "program in the specific." We'll see that a program

can determine the type of an object at execution time and act on that object accordingly.

Interfaces

We introduce interfaces, which are particularly useful for assigning common functionality to possibly unrelated classes, enabling objects of these classes to be processed polymorphically. Objects of classes that **implement** the same interface can respond to all of that interface's method calls.

Interfaces define and standardize how things, such as people and systems, can interact. For example, a radio's controls serve as an interface between radio users and the radio's internal components. The controls allow users to perform a common set of operations (e.g., turn the radio on or off, change the station, adjust the volume and choose between AM and FM). Radios may implement the controls differently (e.g., push buttons, dials and voice commands). The interface specifies *what* operations a radio must permit users to perform but does not specify *how* the operations are performed.[1]

1. You'll see later in this chapter that Java interfaces may define `default` methods that specify *how* operations are performed by default.

Generative AI

1 Prompt genAIs for examples of how Java's inheritance, polymorphism and interfaces help us model real-world things.

9.2 Superclasses and Subclasses

Often, an object of one class *is an* object of another class as well—e.g., a `CarLoan` *is a* `Loan`, as are `HomeImprovementLoan`s and `MortgageLoan`s. Thus, `CarLoan` can be said to inherit from `Loan`. In this context, `Loan` is a superclass, and `CarLoan` is a subclass. A `CarLoan` is a `Loan` type, but it's incorrect to claim that every `Loan` is a `CarLoan`—it could be any loan type.

Examples of Superclasses and Subclasses

The following table lists several simple examples of superclasses

and subclasses. Super-classes tend to be “more general” and subclasses “more specific:”

Superclass	Subclasses
Loan	CarLoan, HomeImprovementLoan, MortgageLoan
Student	GraduateStudent, UndergraduateStudent
Shape	Circle, Triangle, Rectangle, Sphere, Cube
Employee	Faculty, Staff
BankAccount	CheckingAccount, SavingsAccount
Vehicle	Car, Truck, Airplane, Boat

Because every subclass object *is an* object of its superclass, and one superclass can have many subclasses, the set of objects a superclass represents is often larger than the set of objects any of its subclasses represent. For example, the superclass `Vehicle` represents all vehicles, including `Car`s, `Truck`s, `Airplane`s, `Boat`s, etc. By contrast, subclass `Car` rep-resents a smaller, more specific subset of `Vehicle`s.

University Community Member Class Hierarchy

Inheritance relationships form treelike hierarchical structures—a superclass exists in a hierarchical relationship with its subclasses. Let’s consider a sample class hierarchy—also called an **inheritance hierarchy**. A university community can have thousands of members, including employees, students and alumni, as shown in the following UML class diagram. Every class hierarchy begins with class `Object` (in package `java.lang`), which every class directly or indirectly **extends** (or “inherits from”). For just this first diagram, we explicitly show class `Object` at the top of the hierarchy, as all Java classes inherit from it:

`Employee`s are either `Faculty` or `Staff` members. `Faculty` are either `Administrator`s (e.g., deans and department chairpersons) or `Teacher`s. The hierarchy could contain many other classes. For example, `Student`s can be `Graduate` or `Undergraduate` students. `Undergraduate` students can be `Freshmen`, `Sophomore`s, `Junior`s or `Senior`s, and so on.

Each upward-pointing arrow in this class hierarchy represents an *is-a* relationship. As we follow the arrows, we can state, for example, that “an `Employee` is a `CommunityMember`” and “a `Teacher` is a `Faculty` member.” The **direct superclass** is the superclass from which the subclass explicitly inherits. `CommunityMember` is the direct superclass of the classes `Employee`, `Student` and `Alum`. An **indirect**

superclass is any class above the direct superclass in the **class hierarchy**, which defines the inheritance relationships among classes. `CommunityMember` is the indirect superclass of all the classes below `Employee` in the diagram. Starting from the bottom, you can follow the arrows upward and apply the *is-a* relation-ship to the topmost superclass. For example, an `Administrator` *is a* `Faculty` member, *is an* `Employee`, *is a* `CommunityMember` and, of course, *is an* `Object`. Section 9.4 discusses class `Object` in detail.

Java supports **single inheritance**, in which each class derives from one direct super-class. Java does not support multiple inheritance in which a class derives from more than one direct superclass. Later in the chapter, we explain how to use Java interfaces to realize the benefits of multiple inheritance while avoiding associated problems.

Shape Hierarchy

Now consider the following `Shape` inheritance hierarchy:
The hierarchy begins with superclass `Shape`. `Shape`s are either `TwoDimensionalShape`s or `ThreeDimensionalShape`s—the **direct subclasses** of `Shape`. The hierarchy's third level contains specific `TwoDimensionalShape` and `ThreeDimensionalShape` types. Again, we can follow the arrows from the bottom of the diagram upward to the topmost superclass in this class hierarchy to identify several *is-a* relationships. For example, a `Triangle` *is a* `TwoDimensionalShape` and *is a* `Shape`, while a `Sphere` *is a* `ThreeDimensionalShape` and is *a* `Shape`. This hierarchy could contain many other classes. For example, `Ellipse`s and `Trapezoid`s are also `TwoDimensionalShape`s and a `Cylinder` is a `ThreeDimensionalShape`.

Not every class relationship is an inheritance relationship. For example, Chapter 8 discussed the *has-a* (composition) relationship, in which classes have instance variables that reference objects of other classes. Given `Employee`, `BirthDate` and `TelephoneNumber` classes, it's improper to say that an `Employee` *is a* `BirthDate` or that an `Employee` *is a* `TelephoneNumber`. However, it is proper to say that an `Employee` *has a* `BirthDate`, and an `Employee` *has a* `TelephoneNumber`.

It's possible to treat superclass and subclass objects similarly—their commonalities are expressed in the superclass's members. Objects of all classes that extend (i.e., inherit from) a common superclass can be treated as objects of that superclass—such objects have an *isa* relationship with the superclass.

A subclass can customize the methods it inherits from its

superclass. To do this, the subclass method **overrides** (redefines) the superclass method with an implementation appropriate for the subclass.

Generative AI

1 Prompt genAIs to create a text-based `Shape` hierarchy diagram consisting of `TwoDimensionalShapes` and `ThreeDimensionalShapes`. Compare the results to our `Shape` hierarchy diagram.

9.3 Relationship Between Superclasses and Subclasses

We now use an inheritance hierarchy of employee types in a company's payroll application to demonstrate the relationship between superclasses and subclasses:

- The superclass `SalariedEmployee`s are paid a fixed weekly salary.
- The subclass `SalariedCommissionEmployee`s receive a weekly salary plus a percentage of their sales.

We divide our discussion of the relationship between these classes into two examples. The first declares class `SalariedEmployee`, which directly inherits from class `Object` and contains the `private` instance variables `name` and `salary`. The second example declares class `SalariedCommissionEmployee`, which directly inherits from `SalariedEmployee` and adds `private` instance variables `grossSales` and `commissionRate` and corresponding methods. It also customizes (i.e., overrides) two inherited `SalariedEmployee` methods. The software reuse provided by inheritance lets us write less code when developing the new subclass than if we had to write it from scratch. Then, we show how a `SalariedCommissionEmployee` object can use the inherited `SalariedEmployee` `public` methods to manipulate (in a controlled manner) the `private` instance variables inherited from `SalariedEmployee`.

9.3.1 Creating and Using a `SalariedEmployee` Class

Let's examine class `SalariedEmployee` (Fig. 9.1). Line 5 begins the

class declaration and indicates that the class **`extends`** (i.e., inherits from) class **`Object`** (from package `java.lang`). This causes class `SalariedEmployee` to inherit `Object`'s methods—it does not have any fields. If you don't specify which class a new class `extends`, it implicitly `extends Object`. For this reason, you typically will not include "`extends Object`" in your code—we do so in this one example for demonstration purposes.

Click here to view code image

```
1  // Fig. 9.1: SalariedEmployee.java
2  // SalariedEmployee class represents an employee paid a fixed salary.
3  import java.math.BigDecimal;
4
5  public class SalariedEmployee extends Object {
6     private final String name;
7     private BigDecimal salary; // fixed weekly Salary
8
9     // two-argument constructor
10    public SalariedEmployee(String name, BigDecimal salary) {
11       // implicit call to Object's default constructor occurs here
12
13       // if salary is invalid throw exception
14       if (salary.compareTo(BigDecimal.ZERO) < 0) {
15          throw new IllegalArgumentException("salary must be >= 0.0");
16       }
17
18       this.name = name;
19       this.salary = salary;
20    }
21
22    // return name
23    public String getName() {return name;}
24
25    // set salary amount
26    public void setSalary(BigDecimal salary) {
27       if (salary.compareTo(BigDecimal.ZERO) < 0) {
28          throw new IllegalArgumentException("salary must be >= 0.0");
29       }
30
31       this.salary = salary;
32    }
33
34    // return salary amount
35    public BigDecimal getSalary() {return salary;}
36
37    // calculate earnings
38    public BigDecimal calculateEarnings() {return getSalary();}
39
```

```
40      // return String representation of SalariedEmployee object
41      @Override // indicates that this method overrides a superclass method
42      public String toString() {
43         return String.format("name: %s%nsalary: $%s",
44            getName(), getSalary());
45      }
46   }
```

Fig. 9.1 | `SalariedEmployee` class represents an employee paid a fixed salary.

Overview of Class `SalariedEmployee`'s Methods and Instance Variables

The class declares its instance variables as `private` (lines 6–7), so objects of other classes cannot directly access these variables. Class `SalariedEmployee`'s `public` services include:

- a constructor (lines 10–20),
- a `public getName` method (line 23) to *get* the value of the `final` instance variable `name` (line 6),
- `public` methods `setSalary` and `getSalary` (lines 26–35) to *set* and *get* the instance variable `salary` (line 7),
- method `calculateEarnings` (line 38) to calculate a `SalariedEmployee`'s earnings, and
- method `toString` (lines 41–45) to create a `SalariedEmployee`'s `String` representation.

The constructor and `setSalary` each validate their `salary` arguments to ensure they're greater than or equal to 0 before modifying the instance variable `salary`.

Class `SalariedEmployee`'s Constructor

A superclass's constructors are not inherited, but can be called by subclasses. In fact, Java requires each subclass constructor to call its direct superclass's constructor explicitly or implicitly to initialize the inherited superclass instance variables. We discuss the syntax for calling a superclass constructor explicitly in Section 9.3.2.

In this example, `SalariedEmployee`'s constructor implicitly calls class `Object`'s constructor. **If the code does not include an explicit call to the superclass constructor, Java implicitly calls**

the superclass's default or no-argument constructor. The comment in line 11 indicates where Java implicitly calls the superclass `Object`'s default constructor. **Even if a class does not have constructors, the default constructor that the compiler implicitly declares for the class will call the superclass's default or no-argument constructor.**

We did not validate the `name` argument before assigning it to the `name` instance variable (line 18). We could validate it—perhaps to ensure that the name has a reasonable length.

`SalariedEmployee` Member Function `calculateEarnings`

Method `calculateEarnings` (line 38) calls `getSalary` and returns the result. You may wonder why we bother to provide `calculateEarnings` given that we can call `getSalary` directly and get the same result. As you'll see in Section 9.3.2, class `SalariedCommission-Employee` will override `calculateEarnings` to customize how to calculate the earnings for objects of that class. Then, in Section 9.6, we'll show that polymorphism requires methods in each class that have the same signature.

`SalariedEmployee` Member Function `toString`

Method `toString` (lines 41–45) returns a `String` containing a `SalariedEmployee` object's information, which we obtain by calling the class's *get* methods. Method `toString` is one of the methods that every class inherits directly or indirectly from class `Object` (summarized in Section 9.4). As you've seen, the JVM calls `toString` implicitly when an object must be converted to a `String` representation, such as when you output an object as a `String` with `printf` or format an object as a `String` with `String` method `format`. Class `Object`'s `toString` method returns a `String` that includes the object's class name and the object's hashcode (see Section 9.4). This `toString` is primarily a placeholder a subclass can override to specify an appropriate `String` representation for a subclass object's data.

`SalariedEmployee`'s `toString` method (lines 41–45) overrides (redefines) the inherited `Object` version. To override a superclass method, a subclass must declare a method with the same signature (method name, number of parameters, parameter types and order of parameter types) as the superclass method. `Object`'s `toString` method takes no parameters, so `SalariedEmployee` declares

`toString` with no parameters.

`@Override` Annotation

Line 41 uses the optional **`@Override` annotation** to indicate that the following method declaration (i.e., `toString`) should override an existing superclass method. This annotation helps the compiler catch a few common errors. In this example, we intend to override superclass method `toString`, which is spelled with a lowercase "`t`" and an uppercase "`S`." If you inadvertently use a lowercase "`s`," the compiler will flag this as an error because the superclass does not contain a method named `tostring` with a lowercase "`s`." If you didn't use the `@Override` annotation, `tostring` would be an entirely different method that would not be called if a `SalariedEmployee` object were used where a `String` was needed.

Another common overriding error is declaring the wrong number or types of parameters in the subclass method's parameter list. This creates an unintentional overload of the superclass method rather than overriding the existing method. If you then attempt to call the method (with the correct number and types of parameters) on a subclass object, the superclass's version is invoked—potentially leading to subtle logic errors. When the compiler encounters a method declared with `@Override`, it compares the method's signature with the superclass's method signatures. If there isn't an exact match, the compiler issues an error message, such as "method does not override or implement a method from a super-type." You can then correct your method's signature so that it matches one in the super-class.

Though `@Override` is optional, declare overridden methods with it so the compiler can ensure that you defined their signatures correctly. It's always better to find errors at compile time rather than at runtime. For this reason, the `toString` methods in Chapter 8's examples should have been declared with `@Override` as well.

You Cannot Reduce a Method's Visibility in a Subclass

A `public` superclass method must be `public` in each of that class's direct and indirect subclass. It's a compilation error to override a method with a more restricted access modifier:

- a `public` superclass method cannot be `protected`[2] or `private` in a subclass and

- a `protected` superclass method cannot be `private` in a subclass.

2. We discuss `protected` in Section 9.16.

Doing so would break the *is-a* relationship, which requires that all subclass objects be able to respond to method calls made to `public` methods declared in the superclass. If a `public` method could be overridden as a `private` method, the subclass objects would not be able to respond to the same method calls as superclass objects.

Class `SalariedEmployeeTest`

Figure 9.2 tests class `SalariedEmployee`. Lines 9–10 instantiate a `SalariedEmployee` object. Lines 14–15 use `SalariedEmployee`'s *get* methods to retrieve the object's instance-variable values for output. Line 17 invokes the object's `setSalary` method to change the `salary`. Lines 20–23 output the updated `SalariedEmployee`'s `String` representation and earnings. Again, when you output an object using the `%s` format specifier, the JVM implicitly invokes the object's `toString` method to obtain its `String` representation.

Click here to view code image

```
1  // Fig. 9.2: SalariedEmployeeTest.java
2  // SalariedEmployee class test program.
3  import java.math.BigDecimal;
4  import java.math.RoundingMode;
5
6  public class SalariedEmployeeTest {
7     public static void main(String[] args) {
8        // instantiate SalariedEmployee object
9        var employee = new SalariedEmployee(
10          "Sierra Dembo", new BigDecimal("1000.00"));
11
12       // get SalariedEmployee data
13       System.out.println("Employee information obtained by get methods:");
14       System.out.printf("name: %s%nsalary: $%s%n", employee.getName(),
15          employee.getSalary().setScale(2, RoundingMode.HALF_EVEN));
16
17       employee.setSalary(new BigDecimal("1200.00")); // change salary
18
19       // display updated information and calculate and display earnings
20       System.out.printf("%n%s:%n%s%n%nearnings: $%s%n",
21          "Updated employee information obtained by toString", employee,
22          employee.calculateEarnings().setScale(
23             2, RoundingMode.HALF_EVEN));
24    }
```

```
25    }
```

```
Employee information obtained by get methods:
name: Sierra Dembo
salary: $1000.00

Updated employee information obtained by toString:
name: Sierra Dembo
salary: $1200.00

earnings: $1200.00
```

Fig. 9.2 | `CommissionEmployee` class test program.

Generative AI

1 Prompt genAIs to explain why it's a compilation error to override a method with a more restrictive access modifier, but not with a less restrictive access modifier.

9.3.2 Creating a `SalariedEmployee`/`SalariedCommissionEmployee` Inheritance Hierarchy

Now, let's create a `SalariedCommissionEmployee` class (Fig. 9.3) that inherits from class `SalariedEmployee` (Fig. 9.1). A `SalariedCommissionEmployee` object *is a* `SalariedEmployee`, as indicated by line 6's `extends` clause. Inheritance passes on `SalariedEmployee`'s capabilities to the subclass. A `SalariedCommissionEmployee` also has `BigDecimal grossSales` and `commissionRate` instance variables (Fig. 9.3, lines 7–8), which we'll multiply to calculate the commission earned.

Click here to view code image

```
1  // Fig. 9.3: SalariedCommissionEmployee.java
2  // SalariedCommissionEmployee class inherits from SalariedEmployee and
3  // accesses the superclass's private data via inherited public methods.
4  import java.math.BigDecimal;
5
6  public class SalariedCommissionEmployee extends SalariedEmployee {
7     private BigDecimal grossSales; // gross weekly sales
8     private BigDecimal commissionRate; // commission percentage
9
```

```
10      // four-argument constructor
11      public SalariedCommissionEmployee(String name, BigDecimal salary,
12         BigDecimal grossSales, BigDecimal commissionRate) {
13
14         super(name, salary); // call superclass constructor
15
16         // if grossSales is invalid throw exception
17         if (grossSales.compareTo(BigDecimal.ZERO) < 0) {
18            throw new IllegalArgumentException("gross sales must be >= 0.0");
19         }
20
21         // if commissionRate is invalid throw exception
22         if (commissionRate.compareTo(BigDecimal.ZERO) <= 0 ||
23             commissionRate.compareTo(BigDecimal.ONE) >= 0) {
24            throw new IllegalArgumentException(
25               "commission rate must be > 0.0 and < 1.0");
26         }
27
28         this.grossSales = grossSales;
29         this.commissionRate = commissionRate;
30      }
31
32      // set gross sales
33      public void setGrossSales(BigDecimal grossSales) {
34         if (grossSales.compareTo(BigDecimal.ZERO) < 0) {
35            throw new IllegalArgumentException("gross sales must be >= 0.0");
36         }
37
38         this.grossSales = grossSales;
39      }
40
41      // return gross sales
42      public BigDecimal getGrossSales() {return grossSales;}
43
44      // set commission rate
45      public void setCommissionRate(BigDecimal commissionRate) {
46         if (commissionRate.compareTo(BigDecimal.ZERO) <= 0 ||
47             commissionRate.compareTo(BigDecimal.ONE) >= 0) {
48            throw new IllegalArgumentException(
49               "commission rate must be > 0.0 and < 1.0");
50         }
51
52         this.commissionRate = commissionRate;
53      }
54
55      // return commission rate
56      public BigDecimal getCommissionRate() {return commissionRate;}
57
58      // calculate earnings
59      @Override
```

```
60      public BigDecimal calculateEarnings() {
61         return super.calculateEarnings().add(
62            getGrossSales().multiply(getCommissionRate()));
63      }
64
65      // return String representation of SalariedCommissionEmployee
66      @Override
67      public String toString() {
68         return String.format("%s%ngross sales: $%s%ncommission rate: %s",
69            super.toString(), getGrossSales(), getCommissionRate());
70      }
71   }
```

Fig. 9.3 | `SalariedCommissionEmployee` class inherits from `SalariedEmployee` and accesses the superclass's `private` data via inherited `public` methods.

Inheritance

`SalariedCommissionEmployee` inherits all of `SalariedEmployee`'s members, except its constructor, which is specific to the `SalariedEmployee` superclass. Only the superclass's `public` and `protected`[3] members are directly accessible in the subclass. In inheritance, `public` superclass members remain `public` in the subclass, and `protected` ones remain `protected`. There's only one copy of the inherited members. This makes the code easier to maintain, modify and debug because the code related to a `SalariedEmployee` exists only in that class.

3. We discuss `protected` in Section 9.16. There are no `protected` members in this class.

`SalariedCommissionEmployee` Services Provided to Client Code

Class `SalariedCommissionEmployee`'s `public` services (Fig. 9.3) include:

- its own constructor (lines 11–30),
- the methods `setGrossSales`, `getGrossSales`, `setCommissionRate`, `getCommissionRate`, `calculateEarnings` and `toString` (lines 33–70), and
- the `public` methods inherited from `SalariedEmployee`.

The `setGrossSales` (lines 33–39) and `setCommissionRate` (lines 45–53) methods validate their arguments before modifying the instance

variables `grossSales` and `commissionRate`:

- `setGrossSales` ensures that the `grossSales` value is greater than 0, and
- `setCommissionRate` ensures that the `commissionRate` value is greater than 0 and less than 1.

Recall that `BigDecimal`'s `compareTo` method returns a negative value, zero or a positive value if the method's caller is less than, equal to or greater than the method's argument. We compare these return values to 0 to determine whether the `grossSales` are negative (lines 17 and 34) and whether the `commissionRate` is less than or equal to 0 or greater than or equal to 1 (lines 22–23 and 46–47).

Although `SalariedCommissionEmployee`'s source code does not contain the inherited `SalariedEmployee` members, they're nevertheless a part of the subclass. A `SalariedCommissionEmployee` object also contains `SalariedEmployee`'s `private` instance variables, but they're not directly accessible within the subclass. You can access them only via the inherited `SalariedEmployee public` methods (or `protected` methods if there were any).

`SalariedCommissionEmployee`'s Constructor

Each subclass constructor must call a constructor in its direct superclass to initialize the instance variables inherited from that superclass. So, instantiating a subclass object begins a chain of constructor calls. For each new object you create, class `Object`'s constructor is always the last constructor called in the chain.

The constructor (lines 11–30) does this explicitly via **superclass constructor call syntax** (line 14), which consists of keyword **`super`**, followed by a set of parentheses containing the superclass constructor arguments—this initializes the inherited superclass instance variables `name` and `salary`. The explicit superclass constructor call must be the first statement in the constructor's body—this requirement will be relaxed in a future Java version.[4]

4. Java 23's "JEP 482: Flexible Constructor Bodies (Second Preview)" proposes changing this requirement so a subclass constructor can validate its arguments before calling the superclass constructor. This will enable you to prevent unnecessary work in the event that the subclass constructor throws an exception for an invalid argument. See the book's webpage at `https://deitel.com/javafp5` for an example. Accessed September 19, 2024. `https://openjdk.org/jeps/482`.

After calling the superclass constructor, the `SalariedCommissionEmployee` constructor validates the `grossSales` and `commissionRate` arguments and, if valid, assigns them to the corresponding instance variables. When a superclass contains a no-argument constructor, you can use `super()` to call that constructor explicitly, but this is rarely done.

We mentioned in Chapter 8 that you should not call a class's instance methods from its constructor(s). If you follow this advice, you'll avoid subtle problems caused by calling the class's overridable methods either directly or indirectly, which can lead to runtime errors—see Section 9.10 for additional details.

`SalariedCommissionEmployee` Method `calculateEarnings`

`SalariedCommissionEmployee`'s `calculateEarnings` method (lines 59–63) overrides `SalariedEmployee`'s version to calculate a `SalariedCommissionEmployee`'s earnings. Line 61 uses the expression

```
super.calculateEarnings()
```

to calculate the portion of the earnings based on `salary`. The method then adds that value to the commission and returns the total earnings.

Note the syntax used to invoke an overridden superclass method from a subclass—you place the keyword `super` and a dot (`.`) separator before the superclass method name. This method invocation is a good software engineering practice—if a method performs all or some of the actions needed by another method, call that method rather than duplicate its code. This reduces code-maintenance problems.[5]

5. A downside is that this creates tight coupling between superclass and subclasses that can make large-scale systems difficult to modify. We discuss this further in Section 9.13.

When a subclass overrides a superclass method, the subclass version often calls the superclass version to do a portion of the work. Failure to prefix the superclass method name with the keyword `super` and the dot (`.`) separator when calling the superclass's method causes the subclass method to call itself, potentially creating an error called **infinite recursion**, which would eventually cause the method-call stack to overflow—a fatal runtime error. Recursion, used correctly, is a powerful capability discussed in Chapter 22, Recursion and Big *O*.

`SalariedCommissionEmployee` Method `toString`

Similarly, `SalariedCommissionEmployee`'s `toString` method (Fig. 9.3, lines 66–70) overrides `SalariedEmployee`'s `toString` method. The new version returns a `String` containing `SalariedEmployee`'s `String` representation obtained by the expression in line 69:

```
super.toString()
```

followed by the `SalariedCommissionEmployee`'s `grossSales` and `commissionRate`.

Testing Class `SalariedCommissionEmployee`

Figure 9.4 creates the `SalariedCommissionEmployee` object `employee` (lines 9–11). Lines 14–21 output the `employee`'s data by calling its *get* methods to retrieve the object's instance variable values. Lines 24–25 use `setGrossSales` and `setCommissionRate` to change the `employee`'s `grossSales` and `commissionRate` values, respectively. Then, lines 28–31 display the `employee`'s updated information and earnings, calling `toString` (implicitly) to get the `employee`'s `String` representation and `calculateEarnings` to get the `employee`'s earnings.

Click here to view code image

```
1  // Fig. 9.4: SalariedCommissionEmployeeTest.java
2  // SalariedCommissionEmployee test program.
3  import java.math.BigDecimal;
4  import java.math.RoundingMode;
5
6  public class SalariedCommissionEmployeeTest {
7     public static void main(String[] args) {
8        // instantiate SalariedCommissionEmployee object
9        var employee = new SalariedCommissionEmployee("James Davis",
10          new BigDecimal("500.00"), new BigDecimal("10000.00"),
11          new BigDecimal("0.04"));
12
13       // get SalariedCommissionEmployee data
14       System.out.printf("""
15          Employee information obtained by get methods:
16          name: %s
17          salary: $%s
18          gross sales: $%s
19          commission rate: %s
20          """, employee.getName(), employee.getSalary(),
21          employee.getGrossSales(), employee.getCommissionRate());
22
```

```
23          // update gross sales and commission rate
24          employee.setGrossSales(new BigDecimal("8000.00"));
25          employee.setCommissionRate(new BigDecimal("0.1"));
26
27          // display updated information and calculate and display earnings
28          System.out.printf("%n%s:%n%s%n%nearnings: $%s%n",
29             "Updated employee information obtained by toString", employee,
30             employee.calculateEarnings().setScale(
31                2, RoundingMode.HALF_EVEN));
32       }
33    }
```

```
Employee information obtained by get methods:
name: James Davis
salary: $500.00
gross sales: $10000.00
commission rate: 0.04

Updated employee information obtained by toString:
name: James Davis
salary: $500.00
gross sales: $8000.00
commission rate: 0.1

earnings: $1300.00
```

Fig. 9.4 | `SalariedCommissionEmployee` test program.

Notes on Class `SalariedCommissionEmployee`

We could have copied code from class `SalariedEmployee` and pasted it into class `SalariedCommissionEmployee`, then modified class `SalariedCommissionEmployee` to include `grossSales` and `commissionRate` instance variables and methods to manipulate them. This "copy-and-paste" approach would be error-prone and time-consuming. Worse yet, it spreads copies of the same code throughout a system, creating code-maintenance problems—changes to the duplicated code would need to be made in multiple classes.

With inheritance, the instance variables and methods that are the same for all the classes in the hierarchy are declared in a superclass. Changes made to these common features in the superclass are inherited by the subclasses. Without inheritance, changes would need to be made to all the source-code files that contain a copy of the code in question. By using inheritance and by calling superclass methods that hide the superclass's `private` data and ensure its

consistency, we've efficiently and effectively constructed a well-engineered subclass. But can we do better?

Class Design

At the design stage in an object-oriented system, you'll often find that classes share common characteristics. You can identify and "factor out" the common instance variables and methods and place them in a superclass. Then, you can use inheritance to create subclasses that extend these shared characteristics with additional specialized capabilities. Declaring a subclass does not affect its superclass's source code. While inheritance is a powerful tool, experience has shown that it has weaknesses. Section 9.13 discusses why you might prefer composition and interfaces to inheritance.

Generative AI

1 Prompt genAIs, asking if `@Override` is considered to be part of the method it precedes.

9.4 Class `Object`

All classes in Java inherit directly or indirectly from class `Object` (package `java.lang`), so they all inherit the `Object` methods summarized in the following table. We discuss several of these throughout this book (as indicated in the table).

Method	Description
`equals`	This method compares two objects for equality and returns `true` if they're equal and `false` otherwise. The method takes any `Object` as an argument. When objects of a particular class must be compared for equality, the class should override `equals` to compare the contents of the two objects. For the requirements of implementing this method (which include also overriding method `hashCode`), see `https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/lang/Object.html`. The default `equals` implementation uses the `==` operator to determine whether two references refer to the same object in memory.
`hashCode`	Returns an object's hashcode—an `int` value typically used for high-speed storage and retrieval of information in a hashtable data structure (Section 12.10). This method is also called as part

of `Object`'s default `toString` method implementation.

`toString`
This method (Section 9.3.1) returns an object's `String` representation. The default implementation returns the package name and class name of the object's class, followed by the object's `hashCode` value.

`wait`, `notify`, `notifyAll`
Methods `notify`, `notifyAll` and the three overloaded versions of `wait` are related to multithreading.

`getClass`
Every object in Java knows its own type at execution time. Method `getClass` (used in Section 9.8.4) returns an object of class `Class` (package `java.lang`) containing information about the object's type, such as its class name (returned by `Class` method `getName`).

`clone`
This `protected` method, which takes no arguments and returns an `Object` reference, makes a copy of the object on which it's called. The default implementation performs a **shallow copy**—instance-variable values in one object are copied into another object of the same type. Only the references are copied for reference types. A typical overridden `clone` method's implementation would perform a **deep copy** that creates a new object for each reference-type instance variable. Implementing `clone` correctly is difficult. For this reason, its use is discouraged.

9.5 Intro to Polymorphism: Polymorphic Video Game

Suppose we're designing a video game containing `Martian`s, `Venusian`s, `Plutonian`s, `SpaceShip`s and `LaserBeam`s. Each inherits from a `SpaceObject` superclass with the method `draw` and implements this method appropriately for that subclass.

Screen Manager

A screen-manager program maintains an `ArrayList` of `SpaceObject` references to objects of the various classes. The screen manager periodically sends each object the same `draw` message to refresh the screen. Each object responds in its unique way. For example,

- a `Martian` might draw itself in red with the appropriate number of antennae,

- a `SpaceShip` might draw itself as a silver flying saucer and
- a `LaserBeam` might draw itself as a bright red beam across the screen.

The same `draw` message has many forms of results—hence the term polymorphism.

Adding New Classes to the System

A polymorphic screen manager facilitates adding new classes to a system with minimal code modifications. Suppose we want to add `Mercurian` objects to our game. We build a class `Mercurian` that inherits from `SpaceObject` and defines method `draw`, then add the object's reference to the `ArrayList` of `SpaceObject`s. The screen-manager code invokes `draw` the same way for every object the `ArrayList` refers to, regardless of the object's type. So, the new `Mercurian` objects just "plug right in."

Polymorphism enables you to deal in generalities and let the execution-time environment concern itself with the specifics. You can direct objects to behave appropriately without knowing their types as long as they belong to the same class hierarchy and are accessed via a superclass variable.

Polymorphism enhances extensibility by allowing software to operate independently of specific object types. New object types can be integrated into a system without altering its core functionality, as long as they can respond to existing messages. Only code that instantiates new objects must be modified to accommodate new types.

Generative AI

1 Prompt genAIs to explain how Java polymorphism enhances extensibility.

9.6 Demonstrating Polymorphic Behavior

Section 9.3 created a class hierarchy, in which `SalariedCommissionEmployee` inherited from `SalariedEmployee`. The examples in that section manipulated `SalariedEmployee` and

`SalariedCommissionEmployee` objects by using references to them to invoke their methods. We aimed a superclass variable at a superclass object and a subclass variable at a subclass object. These assignments are natural and straightforward, as superclass variables are intended to refer to superclass objects, and subclass variables are intended to refer to subclass objects. Other assignments are possible.

In the next example, we aim a superclass reference at a subclass object. We then show how invoking a method on a subclass object via a superclass reference invokes the subclass functionality. **The referenced object's type, not the variable's type, determines which method is called.** This example demonstrates that **an object of a subclass can be treated as an object of its superclass**, enabling various interesting manipulations. For example, **a program can create an array of superclass variables that refer to objects of many subclass types**. This is allowed because each subclass object is an object of its superclass. We can assign the reference of a `SalariedCommissionEmployee` object to a superclass `SalariedEmployee` variable because a `SalariedCommissionEmployee` *is a* `SalariedEmployee`.

The example in Fig. 9.5 demonstrates three ways to use superclass and subclass variables to store references to superclass and subclass objects. The first two are straightforward—we assign a superclass reference to a superclass variable and a subclass reference to a subclass variable. Then, we demonstrate the relationship between subclasses and super-classes (i.e., the *is-a* relationship) by assigning a subclass reference to a superclass variable. This program uses classes `SalariedEmployee` and `SalariedCommissionEmployee` from Figs. 9.1 and 9.3.

Click here to view code image

```
1  // Fig. 9.5: PolymorphismTest.java
2  // Assigning superclass and subclass references to superclass and
3  // subclass variables.
4  import java.math.BigDecimal;
5  import java.math.RoundingMode;
6
7  public class PolymorphismTest {
8     public static void main(String[] args) {
9        // assign superclass reference to superclass variable
10       var salariedEmployee = new SalariedEmployee("Sierra Dembo",
11          new BigDecimal("1000.00"));
12
13       // assign subclass reference to subclass variable
```

```
14          var salariedCommissionEmployee = new SalariedCommissionEmployee(
15             "James Davis", new BigDecimal("500.00"),
16             new BigDecimal("10000.00"), new BigDecimal("0.04"));
17
18          // invoke toString on superclass object using superclass variable
19          System.out.printf("""
20             Call SalariedEmployee's toString and calculateEarnings
21             methods with superclass reference to superclass object:
22             %s
23             earnings: $%s%n%n""", salariedEmployee.toString(),
24             salariedEmployee.calculateEarnings().setScale(
25                2, RoundingMode.HALF_EVEN));
26
27          // invoke toString on subclass object using subclass variable
28          System.out.printf("""
29             Call SalariedCommissionEmployee's toString and calculateEarnings
30             methods with subclass reference to subclass object:
31             %s
32             earnings: $%s%n%n""", salariedCommissionEmployee.toString(),
33             salariedCommissionEmployee.calculateEarnings().setScale(
34                2, RoundingMode.HALF_EVEN));
35
36          // invoke toString on subclass object using superclass variable
37          SalariedEmployee salariedEmployee2 = salariedCommissionEmployee;
38          System.out.printf("""
39             Call SalariedCommissionEmployee's toString and calculateEarnings
40             methods with superclass reference to subclass object:
41             %s
42             earnings: $%s%n%n""", salariedEmployee2.toString(),
43             salariedEmployee2.calculateEarnings().setScale(
44                2, RoundingMode.HALF_EVEN));
45       }
46    }
```

```
Call SalariedEmployee's toString and calculateEarnings
methods with superclass reference to superclass object:
name: Sierra Dembo
salary: $1000.00
earnings: $1000.00

Call SalariedCommissionEmployee's toString and calculateEarnings
methods with subclass reference to subclass object:
name: James Davis
salary: $500.00
gross sales: $10000.00
commission rate: 0.04
earnings: $900.00

Call SalariedCommissionEmployee's toString and calculateEarnings
methods with superclass reference to subclass object:
```

```
name: James Davis
salary: $500.00
gross sales: $10000.00
commission rate: 0.04
earnings: $900.00
```

Fig. 9.5 | Assigning superclass and subclass references to superclass and subclass variables.

Creating Superclass and Subclass Objects

Lines 10–11 create a `SalariedEmployee` object and assign its reference to a variable of type `SalariedEmployee`. Lines 14–16 create a `SalariedCommissionEmployee` object and assign its reference to a `SalariedCommissionEmployee` variable. These assignments are natural—for example, a `SalariedEmployee` variable's primary purpose is to hold a reference to a `SalariedEmployee` object.

Calling Superclass Methods Via a Superclass Variable

Lines 19–25 use the superclass variable `salariedEmployee` to invoke `SalariedEmployee` methods `toString` (line 23) and `calculateEarnings` (line 24). Because the variable refers to a `SalariedEmployee` superclass object, the superclass versions of both methods are called.

Calling Subclass Methods Via a Subclass Variable

Similarly, lines 28–34 use the subclass variable `salariedCommissionEmployee` to invoke `SalariedCommissionEmployee` methods `toString` (line 32) and `calculateEarnings` (line 33). Because `salariedCommissionEmployee` refers to a `SalariedCommissionEmployee` subclass object, the subclass versions of both methods are called.

Calling Subclass Methods Via a Superclass Variable

Line 37 assigns the `SalariedCommissionEmployee` subclass object referenced by `salaried-CommissionEmployee` to the superclass `SalariedEmployee` variable `salariedEmployee2`. Lines 38–44 use this superclass variable to invoke `SalariedCommissionEmployee` subclass methods `toString` (line 42) and `calculateEarnings` (line 43). **When a superclass variable refers to a subclass object and**

is used to call a method, the subclass version of the method is called. Hence, lines 42–43 call class `SalariedCommissionEmployee`'s methods `toString` and `calculateEarnings`. The Java compiler allows assigning a subclass object to a superclass variable because **an object of a subclass *is an* object of its superclass** (but not vice versa).

When the compiler encounters a method call made through a variable, the compiler determines if the method can be called by checking the variable's class type. The call is compiled if that class contains the proper method declaration (or inherits one). **At execution time, the type of the object to which the variable refers determines the actual method to use.** This process—called **dynamic binding**—enables polymorphic behavior.

Generative AI

1 Prompt genAIs to explain how Java's polymorphism mechanism works.

9.7 `abstract` Classes and Methods

Sometimes, it's useful to declare **abstract classes** for which you never intend to create objects. We refer to them as **abstract superclasses** becausethey're used only as superclasses in inheritance hierarchies. As we'll soon see, these classes cannot be used to instantiate objects because they're incomplete. We demonstrate an abstract class in Section 9.8's code example.

Purpose of `abstract` Classes

An `abstract` class provides a superclass from which other classes can inherit and thus share a common design. In Section 9.2's `Shape` hierarchy, for example, subclasses inherit the notion of what it means to be a `Shape`—perhaps common attributes such as `location`, `fillPattern`, `color` and `borderThickness`, and behaviors such as `draw`, `move`, `resize`, `rotate` and `setColor`.

Classes used to instantiate objects are called **concrete classes**. Such classes implement every method they declare (some implementations might be inherited). For example, we could derive

concrete classes `Circle`, `Square` and `Triangle` from the `abstract` superclass `TwoDimensionalShape`. Similarly, we could derive concrete classes `Sphere`, `Cube` and `Tetrahedron` from the `abstract` superclass `ThreeDimensionalShape`.

`abstract` superclasses are too general to instantiate real objects—they specify only commonality among subclasses. We need to be more specific before we can instantiate objects. For example, if you send the `draw` message to `abstract` class `TwoDimensionalShape`, the class knows that two-dimensional shapes should be drawable, but it does not know what specific shape to draw, so it cannot implement a real `draw` method. **Concrete classes provide the specifics that make it possible to instantiate objects.**

Not all hierarchies contain `abstract` classes. However, you'll often write client code that uses only variables of `abstract` superclass types to reduce the client code's dependencies on a range of subclass types. For example, **you can write a method that receives an abstract superclass type parameter. When called, such a method can receive an object of any concrete class that directly or indirectly `extends` that superclass.**

`abstract` classes sometimes constitute several levels of a hierarchy. For example, In Section 9.2's `Shape` hierarchy, class `Shape` and its direct subclasses `TwoDimensionalShape` and `ThreeDimensionalShape` would each be `abstract` because they are not specific enough to determine the actual shape. The hierarchy's third level declares concrete `TwoDimensionalShape`s(`Circle`, `Square` and `Triangle`) and `ThreeDimensionalShape`s (`Sphere`, `Cube` and `Tetrahedron`).

Declaring an `abstract` Class and `abstract` Methods

You declare a class `abstract` by placing the **`abstract` keyword** before the `class` keyword. An `abstract` class declares common attributes and behaviors (`abstract` and concrete) for the classes in a class hierarchy. An `abstract` class's instance variables and concrete methods are subject to the standard rules of inheritance.

An `abstract` class normally contains one or more **`abstract` methods**, but this is not required. An `abstract` method is an instance method declared `abstract`, as in

Click here to view code image

```
public abstract void draw(); // abstract method
```

`abstract` methods do not provide implementations. A class

containing any `abstract` methods must be explicitly declared `abstract`. Subclasses must declare the "missing piece(s)," if any, to become concrete classes from which you can instantiate objects. So, a concrete subclass of an `abstract` class must provide a concrete implementation for each `abstract` method. Otherwise, the subclass must be declared `abstract`.

Some Methods Cannot Be `abstract`

Constructors and `static` methods cannot be `abstract`. Constructors are not inherited, so an `abstract` constructor could never be implemented. Though non-`private` `static` methods are inherited, they cannot be overridden. An `abstract` method is meant to be overridden for polymorphic processing of objects. A `static` method is associated with a class, not specific objects, so declaring a `static` method `abstract` would not make sense.

Using `abstract` Classes to Declare Variables

Although we cannot instantiate objects of `abstract` superclasses, you'll soon see that **we can use an `abstract` superclass to declare variables that can refer to objects of any concrete class derived from that superclass. We'll use such variables to manipulate subclass objects polymorphically.** You can also use `abstract` superclass names to invoke `static` methods declared in those `abstract` superclasses.

Layered Software Systems

Polymorphism is particularly effective for implementing so-called layered software systems. In operating systems, for example, each physical device type could operate quite differently. Even so, commands to read or write data from and to devices may have a certain uniformity. For each device, the operating system uses software called a device driver to control all communication between the system and the device. The write message sent to a device-driver object needs to be interpreted specifically in the context of that driver and how it manipulates devices of a specific type. However, the write call to one device is no different from the write to any other device in the system—place some number of bytes from memory onto that device.

An object-oriented operating system might use an `abstract` superclass to provide a common "interface" appropriate for all device drivers. Then, subclasses that behave similarly are formed

through inheritance from that `abstract` superclass. The device-driver methods are declared as `abstract` methods in the superclass. The implementations of these `abstract` methods are provided in the concrete subclasses that correspond to the specific types of device drivers. New devices are constantly being developed, often long after the operating system has been released. When you buy a new device, it comes with a device driver provided by the device vendor. The device is immediately operational after you install the driver and connect the device to your computer. The new devices and their drivers "plug right in." This is another elegant example of how polymorphism makes systems extensible.

Generative AI

1 Prompt genAIs, asking why we would want to declare a Java class `abstract` even if it has no `abstract` methods.

9.8 Case Study: Payroll System Using Polymorphism

Let's use an `abstract` class and polymorphism to calculate payroll for two employee types. We create an `Employee` hierarchy to solve the following problem:

> A company pays its employees weekly. The employees are of two types:
>
> - Salaried employees receive a fixed salary regardless of their hours worked.
> - Commission employees receive a percentage of their sales.
>
> For the current pay period, the company has decided to reward salaried employees by raising their salaries by 10%. The company wants to implement a program that performs its payroll calculations polymorphically.

`Employee` Hierarchy

We use the `abstract` class `Employee` to represent the general concept of an employee and define the "interface" to the hierarchy—that is, the methods all `Employee` subclasses must have so a program can invoke those methods on all `Employee` objects. Classes `SalariedEmployee` and `CommissionEmployee` inherit directly from `Employee`. The following UML class diagram shows our polymorphic payroll application's inheritance hierarchy. The `abstract` class name `Employee` is italicized, per UML convention.

The `abstract` superclass `Employee` declares the "interface" to the hierarchy—that is, the methods that a program can invoke on all `Employee` objects throughout the hierarchy. We use the term "interface" here in a general sense to refer to the various ways programs can communicate with objects of any `Employee` subclass. Be careful not to confuse the general notion of an "interface" with the formal notion of a Java interface (Section 9.11). Each employee, regardless of how their earnings are calculated, has a name, so we'll declare the `private` instance variable `name` in the `abstract` superclass `Employee`.

The following diagram shows each class in the hierarchy down the left side and methods `calculateEarnings` and `toString` across the top. For each class, the diagram shows the desired results of each method. We do not list superclass `Employee`'s `getName` method because it's not overridden by any subclasses—it's inherited and used "as is."

The following sections implement the `Employee` class hierarchy. The first declares `abstract` superclass `Employee`. Each of the next two sections implements one of the concrete subclasses. The last section implements a test program that builds objects of all these classes and processes them polymorphically.

9.8.1 `abstract` Superclass `Employee`

Class `Employee` (Fig. 9.6) provides methods `getName`, `toString` and `calculateEarnings`.

Click here to view code image

```
1  // Fig. 9.6: Employee.java
2  // Employee abstract superclass.
3  import java.math.BigDecimal;
4
5  public abstract class Employee {
6     private final String name;
```

```
7
8       // constructor
9       public Employee(String name) {this.name = name;}
10
11      // return name
12      public String getName() {return name;}
13
14      // return String representation of Employee object
15      @Override
16      public String toString() {return String.format("name: %s", getName());}
17
18      // abstract method must be overridden by concrete subclasses
19      public abstract BigDecimal calculateEarnings(); // no implementation
20  }
```

Fig. 9.6 | `Employee` `abstract` superclass.

Class `Employee`'s declaration (Fig. 9.6) includes:

- a constructor (line 9) that receives a `String` to initialize the `final` instance variable `name` (declared at line 6)—the constructor does not validate its parameters in this example,
- a `getName` method to return the `name` (line 12),
- a `toString` method (lines 15–16) to return the `String` representation of an `Employee`, and
- an `abstract` method `calculateEarnings` (line 19)—this will be overridden by each concrete `Employee` subclass.

Each `Employee` subclass will override `toString` to create a `String` representation that contains all of the employee's information.

`abstract` Method `calculateEarnings`

Why did we declare `calculateEarnings` as an `abstract` method? Calculating earnings certainly applies to all employees. However, we cannot calculate the earnings for the general notion of an `Employee`—we first must know the specific `Employee`. Declaring this method `abstract` indicates that each concrete subclass must override `calculateEarnings` with an appropriate implementation.

You cannot create `Employee` objects directly because `Employee` is an `abstract` class. However, due to inheritance, every object of any `Employee` subclass *is an* `Employee` and thus is guaranteed to provide method `calculateEarnings`. So, a program can invoke the method polymorphically on any object that *is an* `Employee` via a superclass `Employee` variable.

In Section 9.8.4, we'll create an array of superclass `Employee` variables, each referencing a subclass object that *is an* `Employee`. The program will iterate through the array, polymorphically calling `calculateEarnings` on each object. The compiler allows calls to method `calculateEarnings` via `Employee` variables because it's declared in class `Employee`—again, the variable's type determines which methods a program can call using that variable.

9.8.2 Concrete Subclass `SalariedEmployee`

Class `SalariedEmployee` (Fig. 9.7) extends class `Employee` (line 5) and overrides `abstract` method `calculateEarnings` (lines 33–34), which makes `SalariedEmployee` a concrete class. The class includes:

- a constructor (lines 9–18) that receives a `name` and a `salary` and passes the name to superclass `Employee`'s constructor (line 10) to initialize `Employee`'s `private` instance variable;
- a `setSalary` method to assign a new nonnegative value to the `salary` instance variable (lines 21–27),
- a `getSalary` method to return `salary`'s value (line 30),
- a `calculateEarnings` method (lines 33–34) that overrides the superclass version to calculate a `SalariedEmployee`'s earnings and
- a `toString` method (lines 37–41) that overrides the superclass version to return a `String` including the information returned by superclass `Employee`'s `toString` and `SalariedEmployee`'s `getSalary` methods.

Click here to view code image

```
1  // Fig. 9.7: SalariedEmployee.java
2  // SalariedEmployee concrete class extends abstract class Employee.
3  import java.math.BigDecimal;
4
5  public class SalariedEmployee extends Employee {
6     private BigDecimal salary; // fixed weekly Salary
7
8     // constructor
9     public SalariedEmployee(String name, BigDecimal salary) {
10       super(name);
11
12       // if salary is invalid throw exception
13       if (salary.compareTo(BigDecimal.ZERO) < 0) {
```

```
14            throw new IllegalArgumentException("salary must be >= 0.0");
15         }
16
17         this.salary = salary;
18      }
19
20      // set salary amount
21      public void setSalary(BigDecimal salary) {
22         if (salary.compareTo(BigDecimal.ZERO) < 0) {
23            throw new IllegalArgumentException("salary must be >= 0.0");
24         }
25
26         this.salary = salary;
27      }
28
29      // return salary amount
30      public BigDecimal getSalary() {return salary;}
31
32      // calculate earnings
33      @Override
34      public BigDecimal calculateEarnings() {return getSalary();}
35
36      // return String representation of SalariedEmployee object
37      @Override // indicates that this method overrides a superclass method
38      public String toString() {
39         return String.format("%s%nsalary: $%s",
40            super.toString(), getSalary());
41      }
42   }
```

Fig. 9.7 | `SalariedEmployee` concrete class extends `abstract` class `Employee`.

Method `calculateEarnings` overrides `Employee`'s `abstract` method to provide a concrete implementation that returns the `SalariedEmployee`'s `salary`. If we do not implement `calculateEarnings`, class `SalariedEmployee` must be declared `abstract`. Of course, we want class `SalariedEmployee` to be concrete in this example.

Method `toString` (lines 37–41) overrides `Employee`'s `toString`. If class `SalariedEmployee` did not override `toString`, it would inherit `Employee`'s version. In that case, class `SalariedEmployee`'s `toString` method would return only the `name`, which does not adequately represent a `SalariedEmployee`.

9.8.3 Concrete Subclass `CommissionEmployee`

Class `CommissionEmployee` (Fig. 9.8) extends class `Employee` (line 5). The class includes:

- a constructor (lines 10–28) that receives a name, a gross sales amount and a commission rate and passes the name to superclass `Employee`'s constructor (line 12) to initialize `Employee`'s `private` instance variable;
- *set* methods (lines 31–37 and 43–51) to assign valid new values to instance variables `grossSales` and `commissionRate`;
- *get* methods (lines 40 and 54) to return the values of `grossSales` and `commissionRate`;
- a `calculateEarnings` method (lines 57–60) that overrides the superclass version to calculate a `CommissionEmployee`'s earnings; and
- a `toString` method (lines 63–67) that overrides the superclass version to return a `String` including the information returned by superclass `Employee`'s `toString` and `CommissionEmployee`'s `getGrossSales` and `getCommissionRate` methods.

Click here to view code image

```
1  // Fig. 9.8: CommissionEmployee.java
2  // CommissionEmployee class extends Employee.
3  import java.math.BigDecimal;
4
5  public class CommissionEmployee extends Employee {
6     private BigDecimal grossSales; // gross weekly sales
7     private BigDecimal commissionRate; // commission percentage
8
9     // constructor
10    public CommissionEmployee(String name,
11       BigDecimal grossSales, BigDecimal commissionRate) {
12       super(name);
13
14       // if grossSales is invalid throw exception
15       if (grossSales.compareTo(BigDecimal.ZERO) < 0) {
16          throw new IllegalArgumentException("gross sales must be >= 0.0");
17       }
18
19       // if commissionRate is invalid throw exception
20       if (commissionRate.compareTo(BigDecimal.ZERO) <= 0 ||
21           commissionRate.compareTo(BigDecimal.ONE) >= 0) {
22          throw new IllegalArgumentException(
23             "commission rate must be > 0.0 and < 1.0");
```

```
24          }
25
26          this.grossSales = grossSales;
27          this.commissionRate = commissionRate;
28       }
29
30       // set gross sales
31       public void setGrossSales(BigDecimal grossSales) {
32          if (grossSales.compareTo(BigDecimal.ZERO) < 0) {
33             throw new IllegalArgumentException("gross sales must be >= 0.0");
34          }
35
36          this.grossSales = grossSales;
37       }
38
39       // return gross sales
40       public BigDecimal getGrossSales() {return grossSales;}
41
42       // set commission rate
43       public void setCommissionRate(BigDecimal commissionRate) {
44          if (commissionRate.compareTo(BigDecimal.ZERO) <= 0 ||
45              commissionRate.compareTo(BigDecimal.ONE) >= 0) {
46             throw new IllegalArgumentException(
47                "commission rate must be > 0.0 and < 1.0");
48          }
49
50          this.commissionRate = commissionRate;
51       }
52
53       // return commission rate
54       public BigDecimal getCommissionRate() {return commissionRate;}
55
56       // calculate earnings
57       @Override
58       public BigDecimal calculateEarnings() {
59          return getGrossSales().multiply(getCommissionRate());
60       }
61
62       // return String representation of CommissionEmployee
63       @Override
64       public String toString() {
65          return String.format("%s%ngross sales: $%s%ncommission rate: %s",
66             super.toString(), getGrossSales(), getCommissionRate());
67       }
68    }
```

Fig. 9.8 | `CommissionEmployee` class extends `Employee`.

9.8.4 Polymorphic Processing, Operator `instanceof` and Downcasting

Figure 9.9 tests our `Employee` hierarchy. Lines 9–12 create a `SalariedEmployee` object and a `CommissionEmployee` object. The program manipulates these objects non-polymorphically, via variables of each object's own type, then polymorphically, using an array of `Employee` variables. While processing the objects polymorphically, the program increases the salary of each `SalariedEmployee` by 10%—this requires determining the object's type at execution time. Finally, the program polymorphically determines and outputs the type of each object in the `Employee` array. Lines 14–21 output each object's `String` representation and earnings non-polymorphically, using the `salariedEmployee` variable to call the `SalariedEmployee` methods and the `commissionEmployee` variable to call the `Commission-Employee` methods. Each object's `toString` method is called implicitly by `printf` when the object is output as a `String` with the `%s` format specifier (lines 16 and 19).

Click here to view code image

```
1  // Fig. 9.9: PayrollSystemTest.java
2  // Employee hierarchy test program.
3  import java.math.BigDecimal;
4  import java.math.RoundingMode;
5
6  public class PayrollSystemTest {
7     public static void main(String[] args) {
8        // create subclass objects
9        var salariedEmployee = new SalariedEmployee("Sierra Dembo",
10          new BigDecimal("1000.00"));
11       var commissionEmployee = new CommissionEmployee("James Davis",
12          new BigDecimal("10000.00"), new BigDecimal("0.06"));
13
14       System.out.println("Employees processed individually:");
15
16       System.out.printf("%s%nearned: $%s%n%n", salariedEmployee,
17          salariedEmployee.calculateEarnings().setScale(
18             2, RoundingMode.HALF_EVEN));
19       System.out.printf("%s%nearned: $%s%n%n", commissionEmployee,
20          commissionEmployee.calculateEarnings().setScale(
21             2, RoundingMode.HALF_EVEN));
22
23       // create and initialize Employee array
24       Employee[] employees = {salariedEmployee, commissionEmployee};
25
26       System.out.println("Employees processed polymorphically:");
```

```
27
28          // generically process each element in array employees
29          for (Employee currentEmployee : employees) {
30             System.out.println(currentEmployee); // invokes toString
31
32             // determine whether element is a SalariedEmployee
33             if (currentEmployee instanceof SalariedEmployee employee) {
34                employee.setSalary(
35                   employee.getSalary().multiply(new BigDecimal("1.10")));
36
37                System.out.printf(
38                   "new salary with 10%% increase is: $%s%n",
39                   employee.getSalary().setScale(2, RoundingMode.HALF_EVEN));
40             }
41
42             System.out.printf("earned: $%s%n%n",
43                currentEmployee.calculateEarnings().setScale(
44                   2, RoundingMode.HALF_EVEN));
45          }
46
47          // get type name of each object in employees array
48          for (int j = 0; j < employees.length; ++j) {
49             System.out.printf("Employee %d is a %s%n", j,
50                employees[j].getClass().getName());
51          }
52       }
53    }
```

```
Employees processed individually:
name: Sierra Dembo
salary: $1000.00
earned: $1000.00

name: James Davis
gross sales: $10000.00
commission rate: 0.06
earned: $600.00

Employees processed polymorphically:
name: Sierra Dembo
salary: $1000.00
new salary with 10% increase is: $1100.00
earned: $1100.00

name: James Davis
gross sales: $10000.00
commission rate: 0.06
earned: $600.00

Employee 0 is a SalariedEmployee
```

```
Employee 1 is a CommissionEmployee
```

Fig. 9.9 | `Employee` hierarchy test program.

Creating the Array of `Employee`s

Line 24 creates the array `employees` and initializes it with the references stored in variables `salariedEmployee` and `commissionEmployee`—the compiler assigns these to `employees` elements 0 and 1, respectively. These assignments are allowed, because a `SalariedEmployee` *is an* `Employee`, and a `CommissionEmployee` *is an* `Employee`. Even though `Employee` is an `abstract` class, we can assign `SalariedEmployee` and `CommissionEmployee` references to superclass `Employee` variables (in this case, the array elements).

Polymorphically Processing `Employee`s

Lines 29–45 iterate through array `employees` and invoke methods `toString` (implicitly; line 30) and `calculateEarnings` (line 43) with `Employee` variable `currentEmployee`, which represents a different `Employee` in the array on each iteration. The output illustrates that a superclass reference can be used to invoke methods of the superclass, but the subclass-specific method implementations are indeed invoked.

The JVM once again uses dynamic binding (also called **late binding**) to resolve all calls to `toString` and `calculateEarnings` at execution time, based on the type of the object to which `currentEmployee` refers. Only the methods of class `Employee` can be called via an `Employee` variable. `Employee`, of course, inherits class `Object`'s methods, so we can also call `Object` methods (like `toString`) via `Employee` variables.

Performing Type-Specific Operations on `CommissionEmployee`s

We perform special processing on `SalariedEmployee` objects. As we encounter these objects at execution time, we increase their salary by 10%. When processing objects polymorphically, we typically do not need to worry about specifics like this. However, to adjust the salary, we must determine the specific type of `Employee` object at execution time.

Line 33 uses the **pattern-matching `instanceof` operator** (Java

16) to determine whether a particular `Employee` *is a* `SalariedEmployee`. The condition in line 33 is true if `currentEmployee` refers to an object that *is a* `SalariedEmployee`—this includes `SalariedEmployee` subclass objects because of the *is-a* relationship a subclass has with its superclass. If the condition is true, the pattern-matching `instanceof` expression automatically converts the reference in `currentEmployee` down from the superclass type `Employee` to the subclass type `SalariedEmployee`, then assigns the reference to the subclass variable `employee`. This conversion is known as a **downcast** and is allowed only if the object `currentEmployee` *is-a* `SalariedEmployee`. The condition at line 33 ensures this is the case. This downcast is required if we're to invoke subclass `SalariedEmployee` methods `getSalary` and `setSalary` on the current `Employee` object—attempting to invoke a subclass-only method directly on a superclass reference is a compilation error.

If the `instanceof` expression in line 33 is `true`, lines 34–39 perform the special processing required for the `SalariedEmployee` object. Using `SalariedEmployee` variable `employee`, lines 34–35 invoke subclass-only methods `getSalary` and `setSalary` to retrieve and update the employee's salary with the 10% raise. Then, lines 37–39 use `get-Salary` to get the new salary and display it.

Calling `calculateEarnings` Polymorphically

Line 43 invokes method `calculateEarnings` on `currentEmployee`, which polymorphically calls the current subclass object's appropriate `calculateEarnings` method. Obtaining the earnings of the `CommissionEmployee` polymorphically produces the same results as obtaining it in line 20. The earnings amount obtained for the `SalariedEmployee` is higher than that in line 17, due to the 10% increase in its base salary.

Getting Each `Employee`'s Class Name

To show that every object knows its own class, lines 48–51 display each `Employee`'s type as a `String`. Every object can access its class's information via the inherited **`Object` method `getClass`**. Method `getClass` returns a reference to an object of type **`Class`** (from package `java.lang`), which contains information about the object's type, such as its class name. Line 50 invokes `getClass` on the current object to get its class. Then, it uses the reference returned by `getClass` to invoke the **`Class` method `getName`** to get the object's class name.

Avoiding Compilation Errors with Downcasting

In this example, we avoided several compilation errors by downcasting an `Employee` variable to a `SalariedEmployee` variable via the pattern-matching `instanceof` expression in line 33. If you attempt to assign the `Employee` variable `currentEmployee` directly to a `SalariedEmployee` variable, you'll receive an "`incompatible types`" compilation error. Assigning the reference stored in a superclass variable to a subclass variable is not allowed because an `Employee` is not a `SalariedEmployee`—**the *is-a* relationship applies only between the subclass and its superclasses, not vice versa.**

Similarly, if lines 35 and 39 were to use superclass variable `currentEmployee` to invoke subclass-only methods `getSalary` and `setSalary`, we'd receive "`cannot find symbol`" compilation errors at these lines. **Attempting to invoke subclass-only methods via a super-class variable is not allowed**—even though lines 35 and 39 execute only if `instanceof` in line 33 returns `true` to indicate that `currentEmployee` holds a reference to a `SalariedEmployee` object. Using a superclass `Employee` variable, we can invoke only the methods in class `Employee`—`getName`, `calculateEarnings` and `toString`. **Although the actual method called depends on the object's type, a variable can be used to invoke only those methods that are members of that variable's type, which the compiler verifies.**

`instanceof` Before Java 16

The pattern-matching `instanceof` operator was finalized in Java 16. Prior to that, line 33's `if` header would have been implemented as follows:

Click here to view code image

```
if (currentEmployee instanceof SalariedEmployee) {
   // downcast Employee reference to SalariedEmployee reference
   SalariedEmployee employee = (SalariedEmployee) currentEmployee;

   // rest of if statement body
}
```

The `if`'s first body statement uses the cast operator—`(SalariedEmployee)`—to convert the superclass `Employee` reference in `currentEmployee` to the subclass variable `employee`. That statement will not execute, however, unless we already know that `currentEmployee` refers to a `SalariedEmployee` object—that is, the

`if`'s `instanceof` condition is `true`. This prevents a `ClassCastException`, which occurs when downcasting if the referenced object does not have an *is-a* relationship with the type specified in the cast. **Although it's allowed, you should generally avoid downcasting.**

Generative AI

1 Prompt genAIs to explain the reasons you should avoid downcasting in Java.
2 Prompt genAIs to explain alternative approaches to downcasting in Java.
3 The polymorphic behavior we discussed in this section is often called **runtime polymorphism** and is implemented using method overriding. Prompt genAIs, asking if Java supports compile-time polymorphism.

9.9 `final` Methods and Classes

You've seen that variables can be declared `final` to indicate that they cannot be modified after they're initialized—such variables represent constant values. It's also possible to declare methods and classes with the `final` modifier.

Final Methods Cannot Be Overridden

A `final`method in a superclass cannot be overridden in a subclass—this guarantees that the `final` method implementation will be used by all direct and indirect subclasses in the hierarchy. **Methods declared `private` are implicitly `final`**—the subclass cannot access its superclass's `private` members, so it's not possible to override them. **Methods declared `static` are also implicitly `final`** because only non-`static` instance methods can be overridden. **Since a `final` method's declaration can never change, calls to `final` methods are resolved at compile time**—known as **static binding**.

Final Classes Cannot Be Superclasses

A **`final`class** cannot be extended, so its methods are implicitly

`final`. `String` is a `final` class. If you were allowed to extend `String`, objects of that subclass could be used wherever `String`s are expected. Since it cannot be extended, programs that use `String`s can rely on the functionality of `String` objects as specified in the Java API. Making the class `final` also prevents programmers from creating subclasses that might bypass security restrictions.

`record` Classes

A `record` class is implicitly a `final` class, so its methods are implicitly `final`. A `record` class's instance variables are also `final`.

If It Can Be Final, It Should Be

In general, if a variable, method or class can be `final`, it should be `final`. This is another example of the principle of least privilege. When we study concurrency in Chapter 18, you'll see that `final` variables make it easier to parallelize your programs for use on today's multi-core processors for improved performance.

Sometimes, declaring classes `final` is important—typically for security reasons. Unless you carefully design a class for extension, you should declare the class as `final` to avoid (often subtle) errors. Like any class, you can reuse a `final` class via composition—for example, our `Employee` class composed a reference to a `String` object, and `String` is a `final` class. For more insights on using `final`, visit

Click here to view code image

```
https://docs.oracle.com/javase/tutorial/java/IandI/final.html
```

Generative AI

1 Prompt genAIs, asking for simple examples of why you might declare a Java class `final` for security purposes.

9.10 Issues with Constructors Calling Instance Methods

In Section 8.10, we chose to duplicate code between class `Time2`'s constructor and `set-Time` methods. We did this because you should not call a class's non-`static` methods from its constructors. To understand why, recall that when you construct a subclass object,

the subclass constructor calls a constructor in its direct superclass before the subclass instance-variable initialization code executes. **If the superclass constructor then calls a method that the subclass overrides, the subclass's version executes before the subclass constructor finishes initializing the subclass object's data.** If the subclass method uses instance variables that have not yet been initialized this could lead to subtle, difficult-to-detect errors.

Let's assume that a constructor and a *set* method perform the same validation for a particular instance variable. How should you handle the common code?

- Duplicating brief validation code in the constructor and the *set* method is a simple way to eliminate the problem we're considering here.
- For lengthier validation code, you can define a `static` validation method—typically a `private static` helper method—then call it from the constructor and the *set* method. It's acceptable to call a `static` method from a constructor because it is not overridable.

It's also acceptable for a constructor to call a `final` or `private` instance method if it does not directly or indirectly call any overridable instance methods.

Generative AI

1 Prompt genAIs, asking for a simple example of a subtle, difficult-to-detect error that could arise from calling an overridable method from a constructor.
2 Prompt genAIs, asking them to discuss potential Java security issues that could arise from calling overridable methods from constructors.

9.11 Creating and Using Interfaces

Our next example (Figs. 9.10–9.13) reexamines Section 9.8's payroll system. Suppose the company wishes to perform several accounting operations in a single accounts-payable application. In addition to

calculating each employee's earnings, the company must also calculate the payments due for invoices (i.e., bills for goods purchased). Though applied to unrelated things—employees and invoices—both operations obtain payment amounts:

- For an employee, the payment amount is the employee's earnings.
- For an invoice, the payment amount is the total cost of the invoice's listed goods.

Can we polymorphically calculate such different things as the payments due for employees and invoices in a single application? Does Java allow unrelated classes to implement a set of common methods (e.g., a method that calculates a payment amount)? Java **interfaces** offer this capability.

Standardizing Interactions

In our car analogy from Section 1.4, a "basic-driving-capabilities" interface consisting of a steering wheel, an accelerator pedal and a brake pedal would enable a driver to tell the car what to do. Once you know how to use this interface for turning, accelerating and braking, you can drive many different cars, even though manufacturers may implement these systems differently. For example, there are many types of braking systems—disc brakes, drum brakes, antilock brakes and more. When you press the brake pedal, your car's brake system type is irrelevant—what matters is that the car slows down.

Software Objects Communicate Via Interfaces

Software objects also communicate via interfaces. A Java interface describes a set of methods that can be called on an object to tell it, for example, to perform some task or return some information. The next example introduces an interface named `Payable` to describe the functionality of any object that must be "capable of being paid" and thus must offer a method to determine the payment amount due.

Using an Interface

To use an interface, a concrete class must

- specify that it **`implements`** the interface and

- provide a concrete implementation of each `abstract` method in the interface.

To specify that a class implements an interface, add the `implements` keyword and the name of the interface to the end of your class declaration's first line, as in:

Click here to view code image

```
public class ClassName extends SuperclassName implements InterfaceName
```

or

Click here to view code image

```
public class ClassName implements InterfaceName
```

InterfaceName may be a comma-separated list of interfaces. **Implementing an interface is like signing a contract with the compiler that states, "I will declare all the `abstract` methods specified by the interface, or I will declare my class `abstract`;"** otherwise, a compilation error occurs.

Tagging Interfaces

A **tagging** (or**marker**)**interface** is an empty interface with no methods or constant values. They're used to add *is-a* relationships to classes. For example, Java has a mechanism called **serialization**, which can convert objects to byte representations (e.g., for storing objects in files or sending them over networks) and can convert those byte representations back to objects (called **deserialization**), using classes `ObjectOutputStream` and `ObjectInput-Stream`. To enable this mechanism to work with your objects, tag them as `Serializable` by adding `implements Serializable` to the end of your class declaration's first line. Then, all the objects of your class have the *is-a* relationship with `Serializable`.

Generative AI

1 Prompt genAIs to compare and contrast Java `abstract` classes and interfaces.
2 Prompt genAIs to explain why many Java developers consider interfaces a more important technology than `abstract` classes.
3 Prompt genAIs, asking if there is discussion in the Java community about removing object serialization from the language.

9.11.1 Developing a `Payable` Hierarchy

To build an application that can determine payments for employees and invoices alike, we first create interface `Payable`, which contains the method `calculatePayment` that returns a `BigDecimal` representing the amount to pay for an object of any class that implements the interface. Method `calculatePayment` is a general-purpose method that can be applied to a broad range of possibly unrelated objects. **When declaring a method in an interface, you should choose a method name that describes the method's purpose in a general manner because the method may be implemented by unrelated classes.**

After declaring `Payable`, we introduce class `Invoice`, which `implements Payable`. We then modify class `Employee` so it, too, `implements Payable`. Then, our program can polymorphically call `calculatePayment` on `Invoice` and `Employee` objects, enabling the required processing of `Invoice`s and `Employee`s for the accounts payable application.

UML Diagram Containing an Interface

The following UML class diagram shows the interface and class hierarchy:
The hierarchy begins with interface `Payable`. The UML distinguishes an interface from classes by placing the word "interface" in guillemets (« and ») above the interface name. The UML expresses the relationship between a class and an interface through a relationship known as **realization**. A class is said to realize (implement) the methods of an interface. A class diagram models a realization as a dashed arrow with a hollow arrowhead pointing from the implementing class to the interface. The preceding diagram indicates that classes `Invoice` and `Employee` each realize interface `Payable`. Class `Employee` appears in italics, indicating that it's an `abstract` class. Concrete class `SalariedEmployee extends Employee`, inheriting its superclass's realization relationship with interface `Payable`.

Generative AI

1 Prompt genAIs to list the most frequently used interfaces in the Java API and give a brief explanation of each.

9.11.2 Interface `Payable`

Interfaces are typically `public` types. Like a `public` class, a `public` interface must be declared in a file with the same name as the interface and the `.java` filename extension (e.g., `Payable.java`). The keyword **`interface`** (line 5 of Fig. 9.10) begins an **interface declaration**. This interface contains the `public abstract` method `calculatePayment`. The method has no parameters, but interface methods can have parameters. Interfaces can also have any number of methods.

Click here to view code image

```
1  // Fig. 9.10: Payable.java
2  // Payable interface declaration.
3  import java.math.BigDecimal;
4
5  public interface Payable {
6     public abstract BigDecimal calculatePayment(); // no implementation
7  }
```

Fig. 9.10 | `Payable` interface declaration.

Originally, an `interface` could contain only constants and `public abstract` methods without implementations. Java 8 and 9 enhanced `interface`s to allow method implementations (Section 9.12). Any `interface` methods you declare without bodies are implicitly `public` and `abstract`, but declaring them `public` and `abstract` explicitly clarifies your intentions—this differentiates the interface's `abstract` methods from the other kinds of interface methods. All fields in an interface are implicitly `public`, `static` and `final`.

9.11.3 Class `Invoice`

Class `Invoice` (Fig. 9.11) represents an invoice for one kind of part. The class declares `private` instance variables `partNumber`, `partDescription`, `quantity` and `pricePerItem` (in lines 6–9). Class `Invoice` also contains a constructor (lines 12–28), *get* methods (lines 31–34) and a `toString` method (lines 37–42) that returns a `String` representation of an `Invoice` object.

Click here to view code image

```
1  // Fig. 9.11: Invoice.java
2  // Invoice class that implements Payable.
```

```
3  import java.math.BigDecimal;
4
5  public class Invoice implements Payable {
6     private final String partNumber;
7     private final String partDescription;
8     private final int quantity;
9     private final BigDecimal pricePerItem;
10
11    // constructor
12    public Invoice(String partNumber, String partDescription, int quantity,
13       BigDecimal pricePerItem) {
14       if (quantity < 0) { // validate quantity
15          throw new IllegalArgumentException("Quantity must be >= 0");
16       }
17
18       // validate pricePerItem
19       if (pricePerItem.compareTo(BigDecimal.ZERO) < 0) {
20          throw new IllegalArgumentException(
21             "Price per item must be >= 0");
22       }
23
24       this.quantity = quantity;
25       this.partNumber = partNumber;
26       this.partDescription = partDescription;
27       this.pricePerItem = pricePerItem;
28    }
29
30    // get methods
31    public String getPartNumber() {return partNumber;}
32    public String getPartDescription() {return partDescription;}
33    public int getQuantity() {return quantity;}
34    public BigDecimal getPricePerItem() {return pricePerItem;}
35
36    // return String representation of Invoice object
37    @Override
38    public String toString() {
39       return String.format("%s: %s (%s)%n%s: %d%n%s: $%s",
40          "part number", getPartNumber(), getPartDescription(),
41          "quantity", getQuantity(), "price per item", getPricePerItem());
42    }
43
44    // method required to carry out contract with interface Payable
45    @Override
46    public BigDecimal calculatePayment() {
47       BigDecimal quantity = new BigDecimal(getQuantity());
48       return quantity.multiply(getPricePerItem()); // calculate total cost
49    }
50 }
```

Fig. 9.11 | `Invoice` class that implements `Payable`.

Line 5 indicates that `Invoice` implements interface `Payable`. Like all classes, `Invoice` also implicitly `extends Object`. Class `Invoice` implements the one `abstract` method in interface `Payable`—method `calculatePayment` in lines 45–49. The method calculates and returns the invoice amount—the product of `quantity` and `pricePerItem`, obtained through the appropriate *get* methods. This method satisfies the implementation requirement for this method in interface `Payable`, so we've fulfilled the interface contract with the compiler.

Generative AI

1 In Fig. 9.11, notice that class `Invoice`'s instance variables are all `final`. Prompt genAIs with `Invoice`'s source code and ask them to refactor the class as a `record` class, keeping the overridden `toString` method.

9.11.4 Modifying Class `Employee` to Implement Interface `Payable`

We now modify Fig. 9.6's `Employee` class to implement interface `Payable` (Fig. 9.12). This class declaration is identical to that of Fig. 9.6 with two exceptions:

- Line 5 of Fig. 9.12 indicates that class `Employee` now implements `Payable`.
- Lines 23–24 implements interface `Payable`'s `calculatePayment` method.

Method `calculatePayment` simply calls `Employee`'s `abstract` `calculateEarnings` method. This might seem strange given that `calculateEarnings` does not have an implementation. However, when we call `calculatePayment` at execution time, it will be for an object of a concrete `Employee` subclass. So, line 24 will polymorphically call that subclass's concrete `calculateEarnings` method, which knows how to calculate earnings for objects of that subclass type.

Click here to view code image

```
1  // Fig. 9.12: Employee.java
2  // Employee abstract superclass that implements interface Payable.
```

```
3  import java.math.BigDecimal;
4
5  public abstract class Employee implements Payable {
6     private final String name;
7
8     // constructor
9     public Employee(String name) {this.name = name;}
10
11    // return name
12    public String getName() {return name;}
13
14    // return String representation of Employee object
15    @Override
16    public String toString() {return String.format("name: %s", getName());}
17
18    // abstract method must be overridden by concrete subclasses
19    public abstract BigDecimal calculateEarnings(); // no implementation
20
21    // implementing calculatePayment here enables the entire Employee
22    // class hierarchy to be used in an app that processes Payables
23    @Override
24    public BigDecimal calculatePayment() {return calculateEarnings();}
25 }
```

Fig. 9.12 | `Employee` `abstract` superclass that implements `interface Payable`.

Subclasses of `Employee` and Interface `Payable`

When a class implements an interface, the same *is-a* relationship as inheritance applies. Class `Employee` implements `Payable`, so an `Employee` *is a* `Payable`. Thus, any object of an `Employee` subclass also *is a* `Payable`. So, if we update the class hierarchy in Section 9.8 with the new `Employee` superclass in Fig. 9.12, then all objects of the `Employee` subclasses `SalariedEmployee` and `CommissionEmployee` would be `Payable` objects.

Just as we can assign the reference of a `SalariedEmployee` subclass object to a super-class `Employee` variable, we can assign the reference of a `SalariedEmployee` object (or any other `Employee` derived-class object) to a `Payable` variable. `Invoice` implements `Payable`, so an `Invoice` object also *is a* `Payable` object, and we can assign the reference of an `Invoice` object to a `Payable` variable.

The *is-a* relationship between superclasses and subclasses, and between interfaces and the classes that implement them, holds when passing an object to a method. When a method parameter receives an argument of a superclass or interface

type, the method polymorphically processes the object it receives as an argument.

Using a superclass reference, we can polymorphically invoke any method declared in the superclass and its superclasses (e.g., class `Object`). Using an interface reference, we can polymorphically invoke any method declared in the interface, its superinterfaces (one interface can extend another) and in class `Object`. Calling methods of class `Object` is allowed via an interface variable because the variable must refer to an object to call a method, and all objects have class `Object`'s methods.

Generative AI

1 Prompt genAIs to give an example of a Java interface being extended.

9.11.5 Using Interface `Payable` to Process `Invoice`s and `Employee`s Polymorphically

`PayableInterfaceTest` (Fig. 9.13) illustrates that interface `Payable` can be used to process a set of `Invoice`s and `Employee`s polymorphically in a single application. Lines 10–15 declare and initialize the four-element array `payableObjects`. Lines 11–12 place the references of `Invoice` objects in `payableObjects`' first two elements. Lines 13–14 place the references of `SalariedEmployee` objects in `payableObjects`' last two elements. The elements can be initialized with `Invoice`s and `SalariedEmployee`s because an `Invoice` *is a* `Payable`, a `SalariedEmployee` *is an* `Employee` and an `Employee` *is a* `Payable`.

Click here to view code image

```
1  // Fig. 9.13: PayableInterfaceTest.java
2  // Payable interface test program processing Invoices and
3  // Employees polymorphically.
4  import java.math.BigDecimal;
5  import java.math.RoundingMode;
6
7  public class PayableInterfaceTest {
8     public static void main(String[] args) {
9        // create four-element Payable array
10       Payable[] payableObjects = {
11          new Invoice("01234", "seat", 2, new BigDecimal("375.00")),
12          new Invoice("56789", "tire", 4, new BigDecimal("79.95")),
```

```
13            new SalariedEmployee("Lorena Silva", new BigDecimal("1200.00")),
14            new SalariedEmployee("Hans Olsen", new BigDecimal("800.00"))
15         };
16
17         System.out.println(
18            "Invoices and Employees processed polymorphically:\n");
19
20         // generically process each element in array payableObjects
21         for (Payable currentPayable : payableObjects) {
22            // output currentPayable and its appropriate payment amount
23            System.out.printf("%s:%n%s %npayment due: $%s%n%n",
24               currentPayable.getClass().getName(),
25               currentPayable.toString(), // could invoke implicitly
26               currentPayable.calculatePayment().setScale(
27                  2, RoundingMode.HALF_EVEN));
28         }
29      }
30   }
```

```
Invoices and Employees processed polymorphically:

Invoice:
part number: 01234 (seat)
quantity: 2
price per item: $375.00
payment due: $750.00

Invoice:
part number: 56789 (tire)
quantity: 4
price per item: $79.95
payment due: $319.80

SalariedEmployee:
name: Lorena Silva
salary: $1200.00
payment due: $1200.00

SalariedEmployee:
name: Hans Olsen
salary: $800.00
payment due: $800.00
```

Fig. 9.13 | `Payable` interface test program processing `Invoice`s and `Employee`s polymorphically.

Lines 21–28 polymorphically process each `Payable` object in `payableObjects`, displaying each object's `String` representation and payment amount. Line 24 displays the class name for each `Payable`

object. Line 25 invokes method `toString` via a `Payable` inter-face reference, even though `toString` is not declared in interface `Payable`. **All references (including those of interface types) refer to objects that extend `Object` and, therefore, have a `toString` method.** Method `toString` also can be invoked implicitly here. Line 26 invokes the `Payable` method `calculatePayment` to obtain the payment amount for each object in `payableObjects`, regardless of the actual type of the object. The output reveals that each of the `toString` and `calculatePayment` method calls invokes the appropriate implementation for each object's type.

9.11.6 Some Common Java API Interfaces

You'll use interfaces extensively when developing Java applications. The Java API contains numerous interfaces, and many Java API methods take interface arguments and return interface values. The following table overviews a few of the more popular interfaces of the Java API that we use in later chapters.

Java API interface	Description
`Iterable`	The enhanced `for` statement can iterate over the elements of objects that implement this interface, such as arrays and `ArrayList`s.
`Comparable`	Java's comparison operators (e.g., `<`, `<=`, `>`, `>=`, `==`, `!=`) allow you to compare primitive values. They cannot be used to compare objects. Interface `Comparable` allows objects of a class that `implements` the interface to be compared to one another, commonly used to order objects in a collection, such as an `ArrayList`. We use `Comparable` and a related interface named `Comparator` in Chapter 12, Generic Collections, and Chapter 13, Generic Classes and Methods: A Deeper Look.
Java Collections API	Java Collections API interfaces enable manipulation of collections, such as an `ArrayList` collection. We discuss many additional collections in Chapter 12, Generic Collections. As you'll see, you often use variables of the Java Collections API's interfaces to refer to and manipulate collections of data. Class `ArrayList` is one of many Java API classes that implement multiple interfaces. It implements interface `Iterable`, so the enhanced `for` statement can iterate over its elements. It also implements interface `List` (Section 12.5), which declares the common methods (such as `add`, `remove` and `contains`) you can call on any object representing a list of items—class `LinkedList` also implements interface `List`.
`Runnable`	Implemented by any class that represents a task to perform.

Objects of such a class are often executed in parallel using a technique called multithreading (discussed in Chapter 18, Concurrency: Platform Threads to Virtual Threads). The interface contains one method, `run`, which specifies the behavior of an object when executed.

GUI event-driven graphical user interfaces (GUIs) daily in your desktop and mobile apps. In your web browser, you might enter the address of a website to visit, or you might click a button to return to a previous site. The browser responds to your interaction and performs the desired task. Your interaction is known as an event, and the code the browser uses to respond to an event is known as an event handler. In Chapter 15, JavaFX Graphical User Interfaces: Part 1, you'll build GUIs with event handlers that respond to user interactions. Event handlers are declared in classes that implement an appropriate event-listener interface. Each event-listener interface specifies one or more methods that must be implemented to respond to user interactions.

`AutoCloseable` implemented by classes that can be used with the `try`-with-resources statement (Chapter 10, Exception Handling: A Deeper Look) to help prevent "resource leaks." We use this interface in Chapter 11, Files, I/O Streams, JSON Serialization & CSV Files, and Chapter 20, Accessing Databases with JDBC and SQLite.

Generative AI

1 Prompt genAIs, asking for real-world uses of inheritance, polymorphism and interfaces with respect to the Java Collections Framework's `ArrayList` class.

9.12 Other Interface Features

This section introduces other interface features that you'll see again in later chapters.

9.12.1 `default` Interface Methods

As we mentioned, interface methods originally could be only `public abstract` methods. This meant that an interface specified *what* operations an implementing class must perform but not *how* the class should perform them.

Java 8 enabled interfaces to include **`public default` methods** with concrete default implementations that specify *how* operations are performed. If a class implements such an interface, the class receives the interface's `default` implementations. The implementing class can override the interface's `default` methods if necessary, but that's not required. To declare a `default` method, place the keyword `default` between `public` and the method's return type and provide a concrete method implementation:

Click here to view code image

```
public default returnType methodName(parameters) {
   statement(s)
}
```

Adding Methods to Existing Interfaces

Before `default` methods, adding methods to an interface would break implementing classes that did not define the new methods—you must declare your class `abstract` if you do not define each of an interface's `abstract` methods. Any class that implements an interface will not break when a `default` method is added. You must recompile the interface when you modify it, but an implementing class simply receives the new `default` method's implementation. So, `default` methods enable you to evolve an existing interface by adding new methods to it without breaking the classes that implement it.

9.12.2 `static` Interface Methods

Chapter 12 introduces the `Collections` class, which contains many `static` helper methods for processing objects that implement Java Collections API interfaces `Collection`, `List`, `Map`, `Set` and more. For example, the `Collections` method `sort` can sort the elements of any object that *is a* `List`—that is, the object's class implements the `List` inter-face. With **`static` interface methods** (also added in Java 8), such helper methods can be declared directly in interfaces rather than separate classes.

9.12.3 Functional Interfaces

Any interface containing only one `abstract` method is a **functional interface**.[6] Functional interfaces are used extensively with Java's

lambda capabilities, which we introduce in Chapter 14, Functional Programming with Lambdas & Streams. Lambdas provide a shorthand notation for implementing functional interfaces. Some functional interfaces that you'll use in this book include:

6. Some people call these SAM (single `abstract` method) interfaces.

- `Comparator` (Chapter 12, Generic Collections)—You'll implement this interface to define a method that compares two objects of a given type to determine whether the first object is less than, equal to or greater than the second.
- `ChangeListener` (Chapter 15, JavaFX Graphical User Interfaces: Part 1)—You'll implement this interface to define a method that's called when the user interacts with a slider graphical user interface control.
- `Runnable` (Chapter 18, Concurrency: Platform Threads to Virtual Threads)—You'll implement this interface to define a task that may run in parallel with other parts of your program.

9.12.4 `private` Interface Methods

As you know, a class's `private` helper methods may be called only by the class's other methods. You can declare **`private` interface methods**, enabling you to define methods with common code used only by the interface's other concrete methods. An interface's `private` instance methods can be called directly (i.e., without an object reference) only by the inter-face's other instance methods. An interface's `private static` methods can be called by any of the interface's instance or `static` methods. Including the `default` keyword in a `private` interface method's declaration is a compilation error—`default` methods must be `public`.

Generative AI

1 Prompt genAIs, asking for simple examples of using `default`, `static` and `private` methods in interfaces.

9.13 Program to an Interface, Not an Implementation[7]

7. The concepts we present in this section are widely discussed in the software engineering community and derived from many sources. "Program to an Interface, Not an Implementation" is defined in Gamma, Erich et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995, 17–18, and further emphasized and discussed in Bloch, Joshua. *Effective Java, Third Edition*. Upper Saddle River, NJ: Addison-Wesley, 2018.

There's much discussion in the software engineering community about the relative merits of composition and inheritance. Many professionals believe inheritance is overused and composition often is more appropriate. This section compares these approaches, then refactors Section 9.8's `Employee` hierarchy using composition and interfaces.

Inheritance-Based Designs

Implementation inheritance in which a subclass `extends` a superclass is primarily used to define closely related classes with many of the same instance variables and method implementations. This kind of inheritance creates **tightly coupled** classes in which the super-class's instance variables and methods are inherited into subclasses. Changes to a superclass directly affect all corresponding subclasses.

Tight coupling can make modifying class hierarchies difficult. Consider modifying Section 9.8's `Employee` hierarchy to support retirement plans. There are many retirement plan types, including 401(k)s and IRAs. We might add an `abstract makeRetirementDe-posit` method to the superclass `Employee`. Then we'd define various subclasses such as `SalariedEmployeeWith401k`, `SalariedEmployeeWithIRA`, `CommissionEmployeeWith-401k`, `CommissionEmployeeWithIRA`, etc., each with an overridden `makeRetirementDe-posit` implementation. This approach quickly proliferates subclasses, making the hierarchy more challenging to implement and maintain.

Rethinking the `Employee` Hierarchy: Composition and Dependency Injection

Over the years, programmers have written numerous papers, articles and blog posts about the problems with tightly coupled class hierarchies like Section 9.8's `Employee` hierarchy. In this example, we'll refactor our `Employee` hierarchy so that the

`Employee`'s compensation model is not "hardwired" into the class hierarchy.[8]

8. We'd like to thank Brian Goetz, Oracle's Java Language Architect, for suggesting the class architecture we use in this section when he reviewed an earlier edition of *Java How to Program*.

To do so, we'll use composition and **dependency injection** in which a class contains a reference to an object that provides a behavior required by objects of the class.[9] In our `Employee` payroll example, that behavior calculates each `Employee`'s earnings. We'll reimplement the `Employee` class using composition—in this example, an `Employee` *has a* reference to a `CompensationModel` object. As you'll see, you can "inject" compensation functionality into an `Employee` object by initializing an `Employee`'s `CompensationModel` reference as you construct the `Employee` or even by setting a new `CompensationModel` at a later time. **Composing a `CompensationModel` reference creates a loosely coupled relationship between an `Employee` and its `CompensationModel`, making it easy to change `CompensationModel`s dynamically without needing to recompile class hierarchies.**

9. "Dependency Injection." Wikipedia. Wikimedia Foundation. Accessed September 19, 2024. https://en.wikipedia.org/wiki/Dependency_injection.

Interfaces Are Best for Flexibility

In this example, we'll declare `CompensationModel` as an interface with an `abstract` method `calculateEarnings`, then declare implementations of `CompensationModel` that specify the ways in which an `Employee` gets paid:

- Class `Salaried implements CompensationModel` for an employee who gets paid a fixed salary.
- Class `Commission implements CompensationModel` for an employee who gets paid a commission based on gross sales.

Class `Employee`'s `calculateEarnings` method will call its composed `Compensation-Model`'s `calculateEarnings` method to determine the `Employee`'s earnings. The new `Employee` class will not be a superclass—to emphasize that, we'll make it `final`.

Generative AI

1 Prompt genAIs, asking why Java implementation inheritance

creates tightly coupled classes.

9.13.1 `CompensationModel` Interface

Figure 9.14 defines the `CompensationModel` interface containing the `abstract` method `calculateEarnings` (line 6). Each class that implements this interface will define the instance variable(s) necessary to calculate an employee's compensation and will override `CompensationModel`'s `calculateEarnings` method to perform the appropriate calculation. **An object of any class that `implements` this interface *is a* `CompensationModel`.**

Click here to view code image

```
1  // Fig. 9.14: CompensationModel.java
2  // CompensationModel interface defining method calculateEarnings.
3  import java.math.BigDecimal;
4
5  public interface CompensationModel {
6     public abstract BigDecimal calculateEarnings();
7  }
```

Fig. 9.14 | `CompensationModel` interface defining method `calculateEarnings`.

9.13.2 An `Employee` Has a `CompensationModel`—Composition and Dependency Injection

Figure 9.15 defines class `Employee`. **This class is declared `final`, so it cannot be a super-class.** Each `Employee` has a reference to its `CompensationModel` implementation (line 7).

Click here to view code image

```
1  // Fig. 9.15: Employee.java
2  // An Employee "has a" CompensationModel.
3  import java.math.BigDecimal;
4
5  public final class Employee {
6     private final String name;
7     private CompensationModel model; // has-a relationship
8
9     // uses "constructor injection" to initialize the model
10    // with a reference to a CompensationModel implementation
11    public Employee(String name, CompensationModel model) {
```

```
12          this.name = name;
13          this.model = model; // constructor injection
14       }
15
16       // set method performs "property injection" to change the
17       // model to a new CompensationModel implementation
18       public void setCompensationModel(CompensationModel model) {
19          this.model = model; // property injection
20       }
21
22       // return the current CompensationModel
23       public CompensationModel getCompensationModel() {return model;}
24
25       // use the CompensationModel to calculate the Employee's earnings
26       public BigDecimal calculateEarnings() {
27          return model.calculateEarnings();
28       }
29
30       // return String representation of Employee object
31       @Override
32       public String toString() {
33          return String.format("%s\n%s", name, model.toString());
34       }
35    }
```

Fig. 9.15 | An `Employee` "has a" `CompensationModel`.

Class `Employee` provides the following constructor and methods:

- The constructor (lines 11–14) initializes the `Employee`'s `name` and aims its `model` reference at an object that implements the `CompensationModel` interface. This is a form of dependency injection known as **constructor injection**. The constructor receives a reference to another object and stores it in (i.e., injects it into) the object being constructed.
- The `setCompensationModel` method (lines 18–20) enables the client code to change an `Employee`'s `CompensationModel`. This is a form of dependency injection known as **property injection**. In Java, a **property** is the term used to describe a pair of *set* and *get* methods that provide controlled access to an instance variable.
- The `getCompensationModel` method (line 23) returns an `Employee`'s current `CompensationModel`.
- The `calculateEarnings` method (lines 26–28) determines the `Employee`'s earnings by calling its `CompensationModel`'s `calculateEarnings` method via the `model` reference.

- The overridden `toString` method (lines 31–34) creates an `Employee`'s `String` representation consisting of the `Employee`'s `name`, followed by the compensation information we get by calling the `CompensationModel` implementation's `toString` method via the `model` reference.

9.13.3 `CompensationModel` Implementations

Next, let's define our `CompensationModel` implementations. Objects of these classes will be injected into `Employee` objects to specify how to calculate their earnings.

`Salaried` Class That Implements `CompensationModel`

A `Salaried` compensation model (Fig. 9.16) defines how to pay an `Employee` who receives a fixed salary. The class implements the `CompensationModel` interface (line 5) and provides:

- instance variable `salary` (line 6) to store the weekly salary,
- a constructor (lines 9–16) to validate and initialize the `salary`,
- a `getSalary` method (line 19) to return the `salary` value,
- a `calculateEarnings` method (lines 22–23) that overrides the superclass version to return the `salary` and
- a `toString` method (lines 26–29) that overrides the superclass version to return a `String` representation of a `Salaried` object for use in this program's outputs.

`Salaried` is declared `final` (line 5) so it may not be used as a superclass.

Click here to view code image

```
1  // Fig. 9.16: Salaried.java
2  // Salaried implements the CompensationModel interface.
3  import java.math.BigDecimal;
4
5  public final class Salaried implements CompensationModel {
6     private BigDecimal salary;
7
8     // constructor
9     public Salaried(BigDecimal salary) {
10       // if salary is invalid throw exception
11       if (salary.compareTo(BigDecimal.ZERO) < 0) {
```

```
12            throw new IllegalArgumentException("salary must be >= 0.0");
13         }
14
15         this.salary = salary;
16      }
17
18      // return the salary
19      public BigDecimal getSalary() {return salary;}
20
21      // override CompensationModel abstract method calculateEarnings
22      @Override
23      public BigDecimal calculateEarnings() {return salary;}
24
25      // override Object method toString
26      @Override
27      public String toString() {
28         return String.format("salary: $%s", salary);
29      }
30   }
```

Fig. 9.16 | `Salaried` implements the `CompensationModel` interface.

`Commission` Class That Implements `CompensationModel`

A `Commission` compensation model (Fig. 9.17) defines how to pay an `Employee` commission based on gross sales. The class implements `CompensationModel` (line 5) and provides:

- instance variables `grossSales` (line 6) and `commissionRate` (line 7) to store the data required to calculate an employee's commission,
- a constructor (lines 10–25) to validate and initialize the instance variables,
- *get* methods (lines 28 and 31) to return the instance variable values,
- a `calculateEarnings` method (lines 34–37) that overrides the superclass version to return the `grossSales` times the `commissionRate` and
- a `toString` method (lines 40–44) that overrides the superclass version to return a `String` representation of a `Commission` object for use in this program's outputs.

`Commission` is declared `final` (line 5) so it may not be used as a superclass.

Click here to view code image

```
1  // Fig. 9.17: Commission.java
2  // Commission implements the CompensationModel interface.
3  import java.math.BigDecimal;
4
5  public final class Commission implements CompensationModel {
6     private BigDecimal grossSales;
7     private BigDecimal commissionRate;
8
9     // constructor
10    public Commission(BigDecimal grossSales, BigDecimal commissionRate) {
11       // if grossSales is invalid throw exception
12       if (grossSales.compareTo(BigDecimal.ZERO) < 0) {
13          throw new IllegalArgumentException("gross sales must be >= 0.0");
14       }
15
16       // if commissionRate is invalid throw exception
17       if (commissionRate.compareTo(BigDecimal.ZERO) <= 0 ||
18           commissionRate.compareTo(BigDecimal.ONE) >= 0) {
19          throw new IllegalArgumentException(
20             "commission rate must be > 0.0 and < 1.0");
21       }
22
23       this.grossSales = grossSales;
24       this.commissionRate = commissionRate;
25    }
26
27    // return the grossSales
28    public BigDecimal getGrossSales() {return grossSales;}
29
30    // return the commissionRate
31    public BigDecimal getCommissionRate() {return commissionRate;}
32
33    // override CompensationModel abstract method calculateEarnings
34    @Override
35    public BigDecimal calculateEarnings() {
36       return grossSales.multiply(commissionRate);
37    }
38
39    // override Object method toString
40    @Override
41    public String toString() {
42       return String.format("gross sales: $%s; commission rate: %s",
43          grossSales, commissionRate);
44    }
45 }
```

Fig. 9.17 | `Commission` implements the `CompensationModel`

interface.

9.13.4 Testing the `CompensationModel` Hierarchy

Next, let's create `Employee` objects and initialize each with an appropriate concrete `CompensationModel` implementation (Fig. 9.18).

Click here to view code image

```
// Fig. 9.18: PayrollSystemTest.java
// Processing Employees with various CompensationModels.
import java.math.BigDecimal;
import java.math.RoundingMode;

public class PayrollSystemTest {
   public static void main(String[] args) {
      // create CompensationModels and Employees
      var salaried = new Salaried(new BigDecimal("1000.00"));
      var salariedEmployee = new Employee("Sierra Dembo", salaried);

      var commission = new Commission(
         new BigDecimal("10000.00"), new BigDecimal("0.06"));
      var commissionEmployee = new Employee("James Davis", commission);

      // create and initialize array of Employees
      Employee[] employees = {salariedEmployee, commissionEmployee};

      // print each Employee's information and earnings
      for (Employee employee : employees) {
         // if Salaried CompensationModel, increase salary by 10%
         if (employee.getCompensationModel() instanceof Salaried s) {
            BigDecimal newSalary =
               s.getSalary().multiply(new BigDecimal("1.10"));

            // inject new CompensationModel
            employee.setCompensationModel(new Salaried(
               newSalary.setScale(2, RoundingMode.HALF_EVEN)));
      }

      System.out.printf("%s%nearned: $%s%n%n", employee,
         employee.calculateEarnings().setScale(
            2, RoundingMode.HALF_EVEN));
      }
   }
}
```

```
Sierra Dembo
salary: $1100.00
earned: $1100.00

James Davis
gross sales: $10000.00; commission rate: 0.06
earned: $600.00
```

Fig. 9.18 | Processing `Employee`s with various `CompensationModel`s.

The `main` method operates as follows:

- Line 9 creates a `Salaried CompensationModel` object (`salaried`).
- Line 10 creates the `Employee salariedEmployee` and injects its `Compensation-Model`, passing `salaried` as the second constructor argument.
- Lines 12–13 create a `Commission CompensationModel` object (`commission`).
- Line 14 creates the `Employee commissionEmployee` and inject its `Compensation-Model`, passing `commission` as the second constructor argument.
- Line 17 creates an array of `Employee`s and initializes it with `salariedEmployee`

  and `commissionEmployee`.
- Lines 20–34 calculate each `Employee`'s earnings polymorphically. A requirement of this application is to give a 10% raise to employees who are paid a fixed weekly salary before calculating their earnings. Lines 22–29 use the pattern-matching `instanceof` operator to determine whether the current `Employee`'s `Compensation-Model` is a `Salaried` object. If so, line 22 converts the `CompensationModel` reference to a `Salaried` reference and assigns it to the variable named `s` in the condition. Lines 23–24 use the variable `s` in the `if`'s body to invoke `Salaried`-specific method `getSalary` to get the current `Employee`'s `salary` then increase it by 10%. Lines 27–28 use the new salary to initialize a new `Salaried CompensationModel`, calling `setCompensationModel` to inject the new `Salaried` object into the current `Employee`. Finally, lines 31–33 display the current `Employee`'s `String` representation and (possibly updated) earnings. Note in the output window that the

`salariedEmployee`'s earnings are indeed 10% higher than its initial value in line 9.

9.13.5 Dependency Injection Design Benefits[10]

10. For more information on dependency injection, see the seminal book: Mark Seemann and Steven van Deursen, *Dependency Injection Principles, Practices, and Patterns* (Manning Publications, 2019).

Flexibility If `CompensationModel`s Change

Declaring the `CompensationModel`s as separate classes that implement the same interface provides flexibility for future changes. Suppose a company adds new ways to pay employees. We can simply define a new class that `implements CompensationModel` and overrides `calculateEarnings` appropriately.

Flexibility If `Employee`s Are Promoted

The interface-based, composition-and-dependency-injection approach used in this example is more flexible than Section 9.8's class hierarchy. In Section 9.8, if an `Employee` were promoted, you'd need to change its object type by creating a new object of the appropriate `Employee` subclass, then moving data into the new object. Using dependency injection, you can simply call `Employee`'s `setCompensationModel` method and inject a reference to a different `CompensationModel` that replaces the existing one. Inheritance is done at compile time, so it does not provide this flexibility.

Flexibility If `Employee`s Acquire New Capabilities

The interface-based composition and dependency-injection approach is also more flexible for enhancing class `Employee`. If we decide to support retirement plans (such as 401(k)s and IRAs), we could say that every `Employee` *has a* `RetirementPlan`. First, we'd define the `RetirementPlan` interface with a `makeRetirementDeposit` method and provide appropriate subclass implementations.

Using interface-based composition and dependency injection, as in this example, requires only small changes to class `Employee` to support `RetirementPlan`s:

- an instance variable that refers to a `RetirementPlan`,

- one more constructor argument to initialize the `RetirementPlan` reference and
- a `setRetirementPlan` method we can call if we ever need to inject a different `Employee` `RetirementPlan`.

9.13.6 `interface`s vs. `abstract` Classes

Interface inheritance may require more work than implementation inheritance because concrete classes must provide both data and implementations of the interface's `abstract` methods, even if they're similar or identical among classes. As you've seen, this approach gives you additional flexibility by eliminating the tight coupling between classes.

`interface`s Are More Flexible Than `abstract` Classes

Given that `interface`s may contain instance methods and `static` methods with implementations, `interface`s are appropriate for almost all cases in which you would have used `abstract` classes previously. With the exception of instance variables, you get the benefits that `abstract` classes provide. Also, a new class can implement any number of interfaces but can extend only one class (`abstract` or concrete).

`interface`s Change Less Frequently Than Implementations

Just as superclasses can change, so can interfaces. If the signature of an interface method changes, all corresponding classes would require modification. Experience has shown that interfaces change much less frequently than implementations.

Generative AI

1 Prompt genAIs on when to prefer interfaces vs. `abstract` classes. **2** Prompt genAIs with Section 9.8's `Employee` class hierarchy code and the `main` application class. Ask them to refactor the code so that an `Employee` has a compensation model, using an interface-based hierarchy of compensation models. Compare the generated code to the code we presented in Section 9.13.

9.14 `sealed` Classes and Interfaces

Java 17 introduced **`sealed` classes** and **`sealed` interfaces**[11] that enable developers to strictly control

11. "JEP 409: Sealed Classes." Accessed September 19, 2024. https://openjdk.org/jeps/409.

- the classes that are allowed to extend a `sealed` class,
- the classes that are allowed to implement a `sealed` interface and
- the interfaces that are allowed to extend a `sealed` interface.

The `sealed` keyword gives class and interface hierarchy designers complete control over the allowed direct subtypes and prevents other developers from extending a hierarchy unless an allowed subtype is explicitly declared **`non-sealed`**. `sealed` also simplifies code maintenance because the hierarchy is not prone to the errors introduced by incorrectly extending it with new classes or interfaces. In addition, `sealed` classes and interfaces streamline pattern-matching `switch` expressions by eliminating the `default` case that typically handles unexpected types. All classes in a `sealed` hierarchy are known in advance, so you simply provide a `case` for each class in the hierarchy.

For this demo, we'll modify the `CompensationModel` hierarchy from Section 9.13 to use a `sealed interface`. Classes `Employee`, `Salaried` and `Commission` are identical, so we do not repeat them here.

Sealing the `CompensationModel` Hierarchy

Figure 9.19 declares the `CompensationModel` interface as `sealed` (line 5). The only classes that can implement this interface are listed in its **`permits` clause**'s comma-separated list (line 5). If we wish to support additional `CompensationModel`s in the future, we must add them to this `permits` clause and recompile. For a `sealed` class, you place the `sealed` key-word before the `class` keyword and follow the class name with a `permits` clause.

Click here to view code image

```
1  // Fig. 9.19: CompensationModel.java
2  // sealed CompensationModel interface defining method calculateEarnings.
3  import java.math.BigDecimal;
4
5  public sealed interface CompensationModel permits Salaried, Commission {
6     public abstract BigDecimal calculateEarnings();
7  }
```

Fig. 9.19 | sealed `CompensationModel` interface defining method `calculateEarnings`.

We mentioned that an interface can extend another interface—this enables you to create an interface that enhances the functionality described by an existing interface. In a `sealed` interface that `extends` another interface, the `permits` clause listing the classes that are allowed to implement the interface must appear after the `extends` clause, as in:

Click here to view code image

```
public sealed interface NewInterface extends SuperInterface
      permits ListOfImplementingClasses {
      // interface members
}
```

In a `sealed` class, the `permits` clause listing the classes that are allowed to extend the class must appear after any `extends` or `implements` clauses, as in:

Click here to view code image

```
public sealed class NewClass extends Superclass implements ListOfInterfaces
   permits ListOfSubclasses {
   // class members
}
```

Note Regarding the `Salaried` and Commission Classes

Classes that extend `sealed` classes or implement `sealed` interfaces must be declared

- `final` so they cannot be extended,
- `sealed` and specify a `permits` clause indicating specific allowed subclasses, or
- `non-sealed` to allow them to be extended without restrictions.

In Figs. 9.16 and 9.17, we declared the `Salaried` and `Commission` classes `final`, so they do not require any modifications for this example.

`switch` Expression in `main`

For this example, we replaced Fig. 9.18's use of an `if` statement and the pattern-matching `instanceof` operator with a pattern-matching `switch` expression that yields a value (Fig. 9.20, lines 23–34) to demonstrate pattern matching with a `sealed` hierarchy. Recall that we introduced pattern-matching `switch` expressions that yield values in Section 5.5. Lines 24 and 33 specify `case`s for each of the `sealed` hierarchy's allowed types. **This `switch` expression does not require a `default` case, as a `sealed` hierarchy cannot have unexpected types.** You'll get compilation errors if you expand the `sealed` hierarchy and forget to add appropriate `case`s for new types. Note in the output window that the `salariedEmployee`'s earnings are indeed 10% higher than its initial value in line 9.

Click here to view code image

```
1  // Fig. 9.20: PayrollSystemTest.java
2  // Processing Employees with various CompensationModels.
3  import java.math.BigDecimal;
4  import java.math.RoundingMode;
5
6  public class PayrollSystemTest {
7     public static void main(String[] args) {
8        // create CompensationModels and Employees
9        var salaried = new Salaried(new BigDecimal("1000.00"));
10       var salariedEmployee = new Employee("Sierra Dembo", salaried);
11
12       var commission = new Commission(
13          new BigDecimal("10000.00"), new BigDecimal("0.06"));
14       var commissionEmployee = new Employee("James Davis", commission);
15
16       // create and initialize array of Employees
17       Employee[] employees = {salariedEmployee, commissionEmployee};
18
19       // print each Employee's information and earnings
20       for (Employee employee : employees) {
21          // if Salaried CompensationModel, increase salary by 10%
22          BigDecimal earnings =
23             switch (employee.getCompensationModel()) {
24                case Salaried s -> {
25                   BigDecimal newSalary =
```

```
                        s.getSalary().multiply(new BigDecimal("1.10"));

                     // inject new CompensationModel
                     employee.setCompensationModel(new Salaried(
                        newSalary.setScale(2, RoundingMode.HALF_EVEN)));
                     yield employee.calculateEarnings();
                  }
                  case Commission c -> c.calculateEarnings();
               };

            System.out.printf("%s%nearned: $%s%n%n",
               employee, earnings.setScale(2, RoundingMode.HALF_EVEN));
         }
      }
   }
```

```
Sierra Dembo
salary: $1100.00
earned: $1100.00

James Davis
gross sales: $10000.00; commission rate: 0.06
earned: $600.00
```

Fig. 9.20 | Processing `Employee`s with various `CompensationModel`s.

Generative AI

1 Prompt genAIs to clarify when `permits` clauses are optional in `sealed` classes and interfaces.

9.15 `private` Constructors

We mentioned previously that constructors are typically declared `public`. Sometimes, it's useful to declare one or more `private` constructors in a class.

Preventing Object Instantiation

You can prevent client code from creating objects of a class by making the class's constructors `private`. For example, consider class `Math`, which contains only `public static` constants and

`public static` methods. Since there's no need to create a `Math` object to use the class's constants and methods, its constructor is `private`.

Sharing Initialization Code in Constructors

One use of a `private` constructor is sharing initialization code among a class's other constructors. You can use delegating constructors (introduced in Section 8.10) to call the `private` constructor with the shared initialization code.

Factory Methods

Another common use of `private` constructors is to force client code to use "factory methods" to create objects. A **factory method** is a `public static` method that creates and initializes an object of a specified type (possibly of the same class), then returns a reference to it. A benefit of this architecture is that the method's return type can be an interface or a superclass (`abstract` or concrete).

Generative AI

1 Prompt genAIs for additional insights on when to declare Java classes with `private` constructors.

9.16 `protected` Members

Chapter 8 discussed access modifiers `public` and `private`:

- A class's `public` members are accessible wherever the program has a reference to an object of that class or one of its subclasses.
- A class's `private` members are accessible only within the class itself.

Access modifier `protected` offers an intermediate access level between `public` and `private`:

- **A superclass's `protected` members can be accessed directly by members of that superclass, by members of**

its subclasses and by members of other classes in the same package.

- **A superclass's `protected` members also have package access.** Recall from Section 8.19 that in a program with multiple classes in the same package, the classes can access each other's package-access members directly through references to objects of those classes or through the class name for `static` members.

So, from most to least restrictive, Java's access modifiers for class members are:

- `private`
- package access (the default when no access specifier is listed)
- `protected`
- `public`

A superclass's `public` and `protected` members retain their access modifiers when they become members of the subclass:

- `public` superclass members become `public` subclass members, and
- `protected` superclass members become `protected` superclass members.

A superclass's `private` members are hidden from subclasses and can be manipulated by the subclass only through the `public` or `protected` methods inherited from the super-class. Subclass methods can refer to `public` and `protected` members inherited from the superclass simply by using the member names.

Declaring `private` instance variables helps you test, debug and correctly modify systems. If a subclass could access its superclass's `private` instance variables, classes that inherit from that subclass could also access them. This would eliminate the benefits of information hiding.

Notes on Using `protected` Instance Variables

Inheriting `protected` instance variables enables subclasses to access the variables directly. In most cases, however, **you should use `private` instance variables to encourage proper software engineering. Your code will be easier to maintain, modify and debug.**

Using `protected` instance variables creates several potential problems. First, the subclass object can set an inherited variable's value directly without using a *set* method. Therefore, a subclass object can assign an invalid value to the variable, possibly leaving the object in an inconsistent state.

Another problem with `protected` instance variables is that subclass methods are more likely to depend on the superclass's data implementation. **In practice, subclasses should depend only on the superclass services (i.e., non-`private` methods) and not the super-class's data implementation.** With `protected` instance variables in the superclass, we may need to modify all the subclasses of the superclass if the superclass implementation changes. Such a class is considered to be **fragile** or **brittle** because a change in the superclass can "break" subclass implementation. **You should be able to change the superclass implementation while still providing the same services to the subclasses.** Of course, if the superclass services change, we must reimplement its subclasses.

A third problem is that a class's **`protected` members are visible to all classes in the same package as the class containing the `protected` members**—this is not always desirable. Declaring superclass instance variables `private` (as opposed to `protected`) enables the superclass implementation of these instance variables to change without affecting subclass implementations.

Use the `protected` access modifier when a superclass should provide a method only to its subclasses and other classes in the same package but not to other clients. Avoid `protected` instance variables. Instead, include non-`private` methods that access `private` instance variables.

Generative AI

1 Prompt genAIs for real-world Java use-cases for `protected` access.

9.17 Wrap-Up

This chapter introduced inheritance—the ability to create classes by acquiring an existing class's members (without copying and pasting the code) and having the ability to embellish them with new capabilities. You learned the notions of superclasses and subclasses and used keyword `extends` to create a subclass that inherits

members from a superclass.

We showed how to use the `@Override` annotation to prevent unintended overloading by indicating that a method overrides a superclass method. You used `super` to access overridden superclass members. You saw how constructors are used in inheritance hierarchies. You studied the methods of class `Object`, the direct or indirect superclass of all Java classes. We discussed the merits of designing classes with composition vs. inheritance.

We introduced polymorphism—the ability to process objects that share the same superclass in a class hierarchy as if they were all objects of the superclass. We discussed how polymorphism makes systems extensible and maintainable, then demonstrated how to use overridden methods to effect polymorphic behavior.

We introduced `abstract` classes, which allow you to provide an appropriate super-class from which other classes can inherit. You learned that an `abstract` class can declare `abstract` methods that each concrete subclass must implement and that a program can use variables of an `abstract` class to invoke the subclasses' implementations of `abstract` methods polymorphically. You saw how to determine an object's type at execution time.

We explained the notions of `final` methods and classes. The chapter discussed declaring and implementing an interface as a way for possibly disparate classes to implement common functionality, enabling objects of those classes to be processed polymorphically. We discussed why constructors should not call overridable methods and that it's OK to call `final`, `static` and `private` methods from constructors.

We introduced `default`, `static` and `private` interface methods. We discussed the purpose of `private` constructors. We introduced the access modifier `protected` and mentioned that subclass methods can directly access inherited `protected` superclass members.

We initially focused on implementation inheritance, then pointed out that decades of experience have shown it can be difficult to maintain and modify tightly coupled implementation-inheritance hierarchies. So, we refactored our `Employee` payroll example to use interface inheritance. In that example, we introduced the composition-and-dependency-injection approach in which a class contains a reference to an object that provides behaviors required by objects of the class. Then, we discussed how this interface-based approach makes the example easier to modify and evolve. Finally, we discussed `sealed` classes and interfaces for controlling which

classes can extend a class or implement an interface.

You've now studied classes, objects, encapsulation, inheritance, polymorphism and interfaces—the essential aspects of object-oriented programming. In the next chapter, we'll discuss exceptions in more depth—exception handling enables more robust programs.

Special Section: Generative AI Exercises

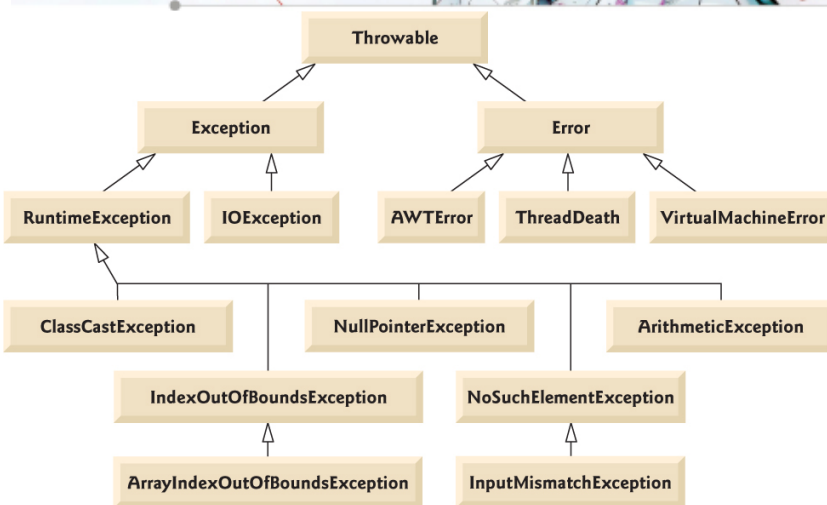
For each of the following questions, try multiple genAIs to determine which performs best. Always verify the results. Keep in mind that they sometimes make mistakes and even hallucinate, creating unreliable information.

9.1 ***(Constructor Injection vs. Property Injection)*** Prompt genAIs to discuss constructor injection vs. property injection and when to use each in Java.

9.2 ***(default Methods)*** Prompt genAIs to discuss the pros and cons of using `default` methods in Java interfaces.

9.3 ***(Runtime Polymorphism vs. Compile-Time Polymorphism)*** Prompt genAIs to discuss runtime vs. compile-time polymorphism in Java and when to use each.

9.4 ***(Composition vs. Inheritance)*** Prompt genAIs to discuss the pros and cons of composition vs. inheritance in Java and to provide examples of when each is more appropriate.


Throwable
Exception
Error
RuntimeException
IOException
AWTError
ThreadDeath
VirtualMachineError
ClassCastException
NullPointerException
ArithmeticException
IndexOutOfBoundsException
NoSuchElementException
ArrayIndexOutOfBoundsException
InputMismatchException

Exception Handling: A Deeper Look

Objectives

In this chapter, you'll:

- Learn why exception handling is an effective mechanism for responding to runtime problems.
- Use `try` blocks to delimit code in which exceptions might occur.
- Use `throw` to indicate that a problem occurred.
- Use `catch` blocks to specify exception handlers.
- Learn when to use exception handling.
- Understand the exception class hierarchy.
- Use the `finally` block to release resources to prevent resource leaks.
- Chain exceptions by catching one exception and throwing another.
- Create custom exceptions.
- Use the debugging feature `assert` to test that a condition is true at a particular point in a method.
- Learn how a `try`-with-resources statement can automatically deallocate a resource when a `try` block terminates.

Outline

10.1 Introduction

10.2 Example: Divide by Zero without Exception Handling
10.3 Example: Handling `ArithmeticException`s and `InputMismatchException`s
10.4 Java Exception Hierarchy
10.5 Checked vs. Unchecked Exceptions
10.6 `finally` Block
10.7 Stack Unwinding and Obtaining Information from an Exception
10.8 Chained Exceptions
10.9 Declaring Custom Exceptions
10.10 Preconditions and Postconditions
10.11 Assertions
10.12 `try`-with-Resources Statement: Automatic Resource Deallocation
10.13 Unnamed Variables in `catch` Handlers
10.14 Wrap-Up
Special Section: Generative AI Exercises

10.1 Introduction

As you've learned in earlier chapters, an exception indicates a problem during a program's execution. Exception handling enables applications to resolve (or handle) exceptions. The features presented in this chapter help you write robust and fault-tolerant programs that can deal with problems and continue executing or terminate gracefully.[1]

1. Java exception handling is based in part on the work of Andrew Koenig and Bjarne Stroustrup—A. Koenig and B. Stroustrup, "Exception Handling for C++ (revised)," *Proceedings of the Usenix C++ Conference*, pp. 149–176, San Francisco, April 1990.

First, we handle an exception that occurs when a method attempts to divide an integer by zero. We discuss when to use exception handling and present a portion of the exception-handling class hierarchy. You'll learn that only subclasses of `Throwable` can

be used with exception handling. We introduce the `try` statement's `finally` block, which executes whether or not an exception occurs. We then demonstrate how to use chained exceptions to add application-specific information to an exception and how to create your own exception types. Next, we introduce preconditions and postconditions, which must be true when your methods are called and when they return, respectively. We then present assertions, which help you debug your code during development. We also discuss catching multiple exceptions with one `catch` handler and the `try`-with-resources statement that automatically releases a resource after it's used in a `try` block.

Exceptions Used Throughout This Book

This chapter focuses on the exception-handling concepts and presents mechanical examples demonstrating key features. Many Java API methods throw exceptions that we handle in our code. The following table shows some of the exception types we've covered, as well as others we'll discuss in later chapters.

Chapter	Examples of exceptions used
Chapter 6	`ArrayIndexOutOfBoundsException`
Chapters 8–9	`IllegalArgumentException`
Chapter 10	`ArithmeticException, InputMismatchException`
Chapter 11	`SecurityException, FileNotFoundException, IOException, ClassNotFoundException, IllegalStateException, FormatterClosedException, NoSuchElementException`
Chapter 12	`ConcurrentModificationException, UnsupportedOperationException, NullPointerException`
Chapter 13	`NoSuchElementException`
Chapter 15	`NumberFormatException`
Chapter 17	`MediaException`
Chapter 18	`InterruptedException, IllegalMonitorStateException, ExecutionException, CancellationException`
Chapter 20	`SQLException`

In industry, organizations develop their own design, coding, testing, debugging and maintenance standards. Companies often have exception-handling standards based on the types of applications they develop, such as real-time systems, high-performance mathematical calculations, big data, network-based distributed systems, artificial intelligence applications, etc.

When to Use Exception Handling

Exception handling is designed to process **synchronous errors**—these occur in a program's regular flow of control as statements execute. Common examples are division by zero, outof-range array indices, arithmetic overflow (i.e., a value outside the representable range of values) and invalid method parameters.

Exception handling provides a single, uniform technique for documenting, detecting and recovering from errors. This helps developers working on large projects understand each other's error-processing code. Exception handling is not meant as an alternate way to specify a program's flow of control or for issues that can be prevented simply by testing a condition before performing a task. Sometimes, you can prevent an exception by validating data first. For example, before you perform integer division, you can ensure that the denominator is not zero, which prevents the `ArithmeticException` that occurs when you divide by zero. Exception handling is not designed to process problems associated with **asynchronous events** (e.g., network message arrivals, mouse clicks and keystrokes), which occur in parallel with and independent of the program's flow of control.

You should incorporate your exception-handling and error-recovery strategy into your system from the inception of the design process. Including these after a system has been implemented can be difficult.

Generative AI

1 Prompt genAIs, asking for a list of 10 commonly thrown Java exceptions and brief explanations of why they might occur.

10.2 Example: Divide by Zero without Exception Handling

First, let's demonstrate what happens when exceptions arise in a program that does not use exception handling. Figure 10.1 prompts the user for two integers and passes them to the `quotient` method, which calculates the integer quotient and returns an `int` result. In

this example, you'll learn that exceptions are **thrown** (i.e., the exception occurs) by a method when it detects a problem and cannot handle it. The first sample execution in Fig. 10.1 shows a successful integer division.

Click here to view code image

```
1  // Fig. 10.1: DivideByZeroNoExceptionHandling.java
2  // Integer division without exception handling.
3  import java.util.Scanner;
4
5  public class DivideByZeroNoExceptionHandling {
6     // demonstrates throwing an exception when a divide-by-zero occurs
7     public static int quotient(int numerator, int denominator) {
8        return numerator / denominator; // possible division by zero
9     }
10
11    public static void main(String[] args) {
12       var input = new Scanner(System.in);
13
14       System.out.print("Enter an integer numerator: ");
15       int numerator = input.nextInt();
16       System.out.print("Enter an integer denominator: ");
17       int denominator = input.nextInt();
18
19       int result = quotient(numerator, denominator);
20       System.out.printf(
21          "%nResult: %d / %d = %d%n", numerator, denominator, result);
22    }
23 }
```

```
Enter an integer numerator: 100
Enter an integer denominator: 7

Result: 100 / 7 = 14
```

```
Enter an integer numerator: 100
Enter an integer denominator: 0
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at DivideByZeroNoExceptionHandling.quotient(
           DivideByZeroNoExceptionHandling.java:8)
        at DivideByZeroNoExceptionHandling.main(
           DivideByZeroNoExceptionHandling.java:19)
```

```
Enter an integer numerator: hello
Exception in thread "main" java.util.InputMismatchException
```

```
        at java.base/java.util.Scanner.throwFor(Scanner.java:964)
        at java.base/java.util.Scanner.next(Scanner.java:1619)
        at java.base/java.util.Scanner.nextInt(Scanner.java:2284)
        at java.base/java.util.Scanner.nextInt(Scanner.java:2238)
        at DivideByZeroNoExceptionHandling.main(
           DivideByZeroNoExceptionHandling.java:15)
```

Fig. 10.1 | Integer division without exception handling.

Stack Trace

In the second execution, the user enters `0` as the denominator. Several lines of information are displayed in response to this invalid input. This **stack trace** includes the exception's package and class name (`java.lang.ArithmeticException`) in a descriptive message that indicates the problem, the line where it occurred and the method-call stack (i.e., the call chain) when the problem occurred. The stack trace shows the path of execution that led to the exception method by method. This helps you debug the program. Even if a problem has not occurred, you can print a stack trace at any time by calling `Thread.dumpStack()`.

Stack Trace for an `ArithmeticException`

The first line specifies that an `ArithmeticException` occurred. The text `/byzero` after the class name indicates that this exception happened due to an attempt to divide (`/`) by zero. **Java does not allow division by zero in integer arithmetic.** When this occurs, Java throws an **`ArithmeticException`**. Such exceptions can arise from several arithmetic problems, so the text `/byzero` provides specific information about the exception's cause to help you understand the problem.

Starting from the stack trace's last line, note that the exception was detected in line 19. Each line of the stack trace contains the class and method names separated by a dot (e.g., `DivideByZeroNoExceptionHandling.main`) followed by the filename and line number (e.g., `DivideByZeroNoExceptionHandling.java:19`) where the problem occurred in that method. Moving up the stack trace, note that method main called method `quotient` where the exception occurred in line 8. The stack trace's top line number indicates the exception's **throw point**—the initial point at which the exception occurred.

Side Note Regarding Floating-Point Arithmetic

Java allows division by zero with floating-point values, resulting in positive or negative infinity, which is displayed as `"Infinity"` or `"-Infinity"`. If you divide 0.0 by 0.0, the result is **`NaN` (not a number)**—a floating-point value displayed as `"NaN"`. To compare a floating-point value to `NaN`, use `Float.isNaN(`*floatValue*`)` or `Double.isNaN(`*doubleValue*`)`. Each returns a `boolean` value indicating whether the argument value is `NaN`. Classes `Float` and `Double` are in package `java.lang`.

Stack Trace for an `InputMismatchException`

In Fig. 10.1's third execution, the user entered `"hello"` as the numerator, resulting in an `InputMismatchException` (package `java.util`). Our prior examples that input numeric values assumed the user would enter a proper integer value. Users sometimes enter erroneous noninteger values. An **`InputMismatchException`** occurs when `Scanner` method `nextInt` receives a `String` that does not represent a valid integer. Starting from the end of the stack trace, note that the exception was detected in line 15. Moving up the stack trace, we notice that the exception occurred in `Scanner`'s method `nextInt`.

Program Termination

In Fig. 10.1, when exceptions occur and Java displays stack traces, the program exits. This does not always happen—sometimes, a program can continue executing. In such cases, the program may produce unexpected results. For example, a graphical user interface (GUI) application will often continue executing. In Fig. 10.1, two exception types were detected in `main`. In the next example, we'll handle these exceptions so that you can enable the program to run to normal completion.

Generative AI

1 Prompt genAIs, asking why exception handling is important for preventing unexpected program termination.

10.3 Example: Handling `ArithmeticException`s and `InputMismatchException`s

Figure 10.2 modifies the previous program, using exception handling to deal with any `ArithmeticException`s and `InputMismatchException`s that arise. The program still prompts the user to enter two integers and passes them to method `quotient`, which calculates the quotient and returns an `int` result. If the user makes an input mistake, this program catches and handles the exception, allowing the user to re-enter the input.

Click here to view code image

```
1  // Fig. 10.2: DivideByZeroWithExceptionHandling.java
2  // Handling ArithmeticExceptions and InputMismatchExceptions.
3  import java.util.InputMismatchException;
4  import java.util.Scanner;
5
6  public class DivideByZeroWithExceptionHandling {
7     // demonstrates throwing an exception when a divide-by-zero occurs
8     public static int quotient(int numerator, int denominator)
9        throws ArithmeticException {
10       return numerator / denominator; // possible division by zero
11    }
12
13    public static void main(String[] args) {
14       var input = new Scanner(System.in);
15       boolean continueLoop = true; // determines if more input is needed
16
17       while (continueLoop) {
18          try { // read two numbers and calculate quotient
19             System.out.print("Enter an integer numerator: ");
20             int numerator = input.nextInt();
21             System.out.print("Enter an integer denominator: ");
22             int denominator = input.nextInt();
23
24             int result = quotient(numerator, denominator);
25             System.out.printf("%nResult: %d / %d = %d%n", numerator,
26                denominator, result);
27             continueLoop = false; // input successful; end looping
28          }
29          catch (InputMismatchException inputMismatchException) {
30             System.err.printf("%nException: %s%n",
31                inputMismatchException);
32             input.nextLine(); // discard input so user can try again
33             System.out.printf(
34                "You must enter integers. Try again.%n%n");
35          }
36          catch (ArithmeticException arithmeticException) {
37             System.err.printf("%nException: %s%n", arithmeticException);
```

```
38                System.out.printf(
39                   "Zero is an invalid denominator. Try again.%n%n");
40             }
41          }
42       }
43    }
```

```
Enter an integer numerator: hello

Exception: java.util.InputMismatchException
You must enter integers. Try again.

Enter an integer numerator: 100
Enter an integer denominator: 0

Exception: java.lang.ArithmeticException: / by zero
Zero is an invalid denominator. Try again.

Enter an integer numerator: 100
Enter an integer denominator: 7

Result: 100 / 7 = 14
```

Fig. 10.2 | Handling `ArithmeticException`s and `InputMismatchException`s.

Lines 17–41 repeatedly ask the user for input until valid inputs are received. In the sample execution:

- First, the user enters `"hello"` when an integer is expected. An `InputMismatch-Exception` occurs, and the program asks the user to try again.
- Next, the user enters a valid numerator (`100`) but a denominator of `0`. An `ArithmeticException` exception occurs, and the program asks the user to try again.
- Finally, the user enters a proper numerator and a non-zero denominator, so the program performs the division, displays the result and completes successfully.

Enclosing Code in a `try` Block

Lines 18–28 contain a **`try` block**, which **encloses the code that might `throw` an exception and the code that should not execute if an exception occurs**—if an exception occurs, the remaining code in the `try` block will be skipped. A `try` block consists of the keyword `try` followed by a block of code enclosed in curly braces.

The statements that read the integers from the keyboard (lines 20 and 22) each use method `nextInt`, which throws an `InputMismatchException` if the value read in is not an integer.

The division that can cause an `ArithmeticException` is not performed in the `try` block. Rather, the call to method `quotient` (line 24) in the `try` block invokes the code that attempts the division (line 10), and the JVM throws an `ArithmeticException` object when the denominator is zero. Exceptions may surface from

- code in a `try` block,
- method calls initiated by code in a `try` block or
- the Java Virtual Machine as it executes Java bytecodes.

Catching Exceptions

This example's `try` block is followed by two `catch` blocks:

- one handles an `InputMismatchException` (lines 29–35) and
- one handles an `ArithmeticException` (lines 36–40).

A `catch` block (also called a **`catch` clause** or **exception handler**) catches and handles an exception thrown from the corresponding `try`. The `catch` keyword is followed by

- required parentheses containing an **exception parameter** identifying the exception type the handler can process and
- a block of code enclosed in curly braces.

At least one `catch` block—or a `finally` block (Section 10.6)—must immediately follow the `try` block. We prefer to name our exception parameters (`inputMismatchException` and `arithmeticException`) based on their types. Many Java programmers use the letter `e` as the name of their exception parameters.

When an exception occurs in a `try` block, the first `catch` whose type matches the thrown exception's type executes. The `catch`'s type matches if it is the thrown exception's type or is a direct or indirect superclass of it. The exception parameter's name enables the `catch` block to interact with a caught exception object (as in lines 30–31 and 37). We used the **`System.err` (standard error stream) object** here to output error messages. Like `System.out`, `System.err`'s print methods display data to the command prompt by default.

If lines 20 or 22 cause an `InputMismatchException`, the `try` block terminates (so method `quotient` is not called), and execution continues with the `catch` block at lines 29–35. Because an `InputMismatchException` occurred, the call to method `nextInt` never successfully read the user's invalid input—so line 32 reads and discards that input with a call to `Scanner` method `nextLine`. Then, lines 33–34 display an error message and ask the user to try again. We'll discuss the `catch` block for `ArithmeticException`s shortly.

When either `catch` block terminates, `main` continues after the last `catch` block, returning to the top of the loop to re-prompt the user for input. We'll soon take a deeper look at how this flow of control works in exception handling.

Multi-`catch`

It's common for a `try` block to be followed by several `catch` blocks to handle various exceptions. If several `catch` blocks' bodies are identical and there are no *is-a* relationships among the exceptions being caught, you can use the **multi-`catch`** feature to catch those exceptions in a single `catch` handler and perform the same task. The syntax for a multi-*`catch`* is:

Click here to view code image

```
catch (Type1 | Type2 | Type3 e)
```

Each exception type is separated from the next with a vertical bar (|). The preceding line of code indicates that any of the types (or their subclasses) can be caught in the exception handler. Any number of exception types can be specified in a multi-`catch`.

Uncaught Exceptions

An **uncaught exception** is one for which there are no matching `catch` blocks. There were uncaught exceptions in the second and third outputs of Fig. 10.1. Recall that when exceptions occurred in that example, the program terminated immediately after displaying the exception's stack trace.

Uncaught exceptions do not always result in program termination. **Java uses a "multithreaded" model of program execution—each thread is a concurrent activity. One program can have many threads. If a program has only one thread, an uncaught exception will cause the program to terminate. If a**

program has multiple threads, an uncaught exception will terminate only the thread in which the exception occurred. Some threads may rely on others. If one thread terminates due to an uncaught exception, there may be adverse effects on the program's other threads. Chapter 18, Concurrency: Platform Threads to Virtual Threads, discusses these issues in depth.

Termination Model of Exception Handling

In Fig. 10.2, the first `catch` block catches `InputMismatchException`s, and the second `catch` block catches `ArithmeticException`s. After an exception is handled, program control does not return to the throw point because the `try` block has expired. Rather, control resumes after the last `catch`. This is known as the **termination model of exception handling**. Some languages use the **resumption model of exception handling**, in which, after an exception is handled, control resumes just after the throw point.

After executing a `catch` block, this program's flow of control proceeds to the first statement after the last `catch` block (line 41). In this case, the program reaches the `while` loop's closing brace, so control returns to the loop's condition. The condition is still `true`, so the program prompts the user for input again. This loop executes until valid input is entered. At that point, line 27 assigns `false` to `continueLoop`, then the `try` block terminates. When no exceptions are thrown in the `try` block, Java skips the `catch` blocks and control continues with the first statement after the `catch` blocks.[2] Now the loop's condition is `false`, and `main` ends.

2. We'll present another possibility when we discuss the `finally` block in Section 10.6.

The `try` block and its corresponding `catch` and/or `finally` blocks form a `try` statement. Do not confuse the terms "`try` block" and "`try` statement"—the latter includes the `try` block and the following `catch` blocks and/or `finally` block.

As with any other block of code, when a `try` block terminates, local variables declared in the block go out of scope and are no longer accessible; thus, a `try` block's local variables are not accessible in the corresponding `catch` blocks. When a `catch` block terminates, local variables declared within that block (including its exception parameter) also go out of scope.

`throws` Clause

In method `quotient` (lines 8–11), the **`throws` clause** (line 9) specifies the exceptions the method might throw if problems occur. This clause, which must appear after the method's parameter list and before the body, contains a comma-separated list of exception types. Such exceptions might be thrown by statements in the method's body or methods called from there. We've added the `throws` clause to this method to indicate to its callers that this method might throw an `ArithmeticException`. Some exception types, such as `ArithmeticException`, are not required to be listed in the `throws` clause. For those that are, the method can throw exceptions that have the *is-a* relationship with the classes listed in the `throws` clause. You'll learn more about this in Section 10.4.

Read a method's online API documentation before using it in a program. The documentation specifies exceptions thrown by the method (if any) and indicates why such exceptions might occur. Next, read the online API documentation for the specified exception classes, which typically lists why such exceptions occur. Finally, handle those exceptions in your program.

Method `quotient` Failing Due to Division By Zero

When line 10 executes, if the `denominator` is zero, the JVM throws an `ArithmeticException` object. This object will be caught by the `catch` block at lines 36–40, which displays the exception's basic information returned by its `toString` method. Then, the program asks the user to try again.

When `quotient` throws an `ArithmeticException`, it terminates immediately and does not return a value. Its local variables go out of scope. If `quotient` contained local variables that were references to objects and there were no other references to those objects, the objects would be marked for garbage collection. In this program, when an `Arithmetic-Exception` occurs, the `try` block from which `quotient` was called terminates before lines 25–27 can execute. Here, too, the `try` block's local variables go out of scope.

Method quotient Successfully Executing

If the `denominator` is not zero, `quotient` performs the division and returns the result to the line in the `try` block that called `quotient` (line 24). Lines 25–26 display the result, and line 27 sets `continueLoop` to `false`. In this case, the `try` block completes successfully, so the program skips the `catch` blocks, the condition at line 17 fails and `main` completes execution normally.

Generative AI

1 Prompt genAIs, asking for real-world examples of when to use exception handling versus input validation.
2 Prompt genAIs, asking why you might not want to `catch` an exception in a Java program.

10.4 Java Exception Hierarchy

All Java exception classes inherit directly or indirectly from **class `Exception`**, forming an inheritance hierarchy. You can extend this hierarchy with your own exception classes.

The following UML class diagram shows a small portion of the inheritance hierarchy for **class `Throwable`** (a subclass of `Object`), which is the superclass of class `Exception`—only `Throwable` objects can be used with the exception-handling mechanism:

Class `Throwable` has two direct subclasses—`Exception` and `Error`:

- Class `Exception` and its subclasses—for example, `RuntimeException` (package `java.lang`) and `IOException` (package `java.io`)—represent exceptional situations that can occur in a Java program and that can be caught by the application.
- **Class `Error`** and its subclasses represent abnormal situations that happen in the JVM. Most `Error`s happen infrequently and should not be caught by applications—it's usually not possible for applications to recover from them.

The Java exception hierarchy contains hundreds of classes. Information about them can be found throughout the Java API. You can view `Throwable`'s documentation at

Click here to view code image

```
https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/
   lang/Throwable.html
```

From there, you can look at this class's subclasses for more information about Java's `Exception`s and `Error`s.

Catching Subclass Exceptions

If a `catch` handler is written to catch superclass exception objects, it will also catch all objects of that exception class's subclasses. This enables a `catch` block to handle related exceptions polymorphically. You should `catch` each subclass individually if they require different processing.

Only the First Matching `catch` Executes

If multiple `catch` blocks match a particular exception type—e.g., one catches a specific exception class, and another catches its superclass—only the first matching `catch` block executes. For example, we could follow a `catch` block for `ArithmeticException` with a `catch` block for `Exception`—both match `ArithmeticException`s, but only the first matching `catch` would execute. Placing a `catch` block for a superclass exception type before other `catch` blocks that catch subclass exception types would prevent those `catch` blocks from executing, so a compilation error occurs.

Catching subclass types individually is subject to error if you forget to test for one or more of the subclass types explicitly. Catching the superclass guarantees that objects of all subclasses will be caught. Positioning a `catch` block for the superclass type after all other subclass `catch` blocks ensures that all subclass exceptions are eventually caught.

In industry, throwing or catching type `Exception` is discouraged —we use it in this chapter to demonstrate exception-handling mechanics. In subsequent chapters, we generally `throw` and `catch` more specific exception types.

Generative AI

1 Prompt genAIs, asking for a list of common `Error` subclasses and the reasons they might occur.
2 Prompt genAIs, asking why you should not `throw` and `catch` type `Exception`.

10.5 Checked vs. Unchecked Exceptions

Java distinguishes between two categories of exceptions—**checked** and **unchecked**. Understanding the distinction between them will help you write more robust code. The Java compiler enforces special requirements for checked exceptions.

`RuntimeException`s Are Unchecked Exceptions

An exception class's location in the Java `Exception` class hierarchy (Section 10.4) determines whether it's checked or unchecked. Every direct or indirect subclass of **`RuntimeException`** (package `java.lang`) is an unchecked exception. These are typically caused by programming errors. Several common unchecked exceptions include:

- `ArithmeticException`s (shown in Figs. 10.1 and 10.2) occur when you attempt to divide by zero in integer arithmetic. To avoid these, ensure that the denominator is not zero before the calculation.
- `ArrayIndexOutOfBoundsException`s (Chapter 6) occur when you try to access array elements outside the array's bounds. To avoid these, ensure that array indices are greater than or equal to 0 and less than the array's `length`.
- `IllegalArgumentException`s are often thrown by constructors and methods when they receive incorrect arguments (as we did in Chapters 8 and 9). You can avoid these by ensuring you pass correct arguments to methods.
- `NullPointerException`s occur when you attempt to call a method via a variable that contains `null`—that is, the variable does not refer to an object. You can avoid these by ensuring that a reference is not `null` before you use it to call a method.
- `StringIndexOutOfBoundsException`s occur when you try to access a character outside a `String`'s bounds. You can avoid these by ensuring that your indices are always greater than or equal to 0 and less than the value returned by `String`'s `length` method.

Errors are Unchecked

Direct and indirect subclasses of `Error` are unchecked by the compiler—they represent critical errors from which programs cannot recover. For example, `VirtualMachineError`'s

documentation[3] says this error is "thrown to indicate that the Java Virtual Machine is broken or has run out of resources necessary for it to continue operating." If this error occurs, there's nothing your program can do, so you should not `catch` it.

3. "Class `VirtualMachineError`." Accessed September 22, 2024. https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/lang/VirtualMachineError.html.

Checked Exceptions

All classes that inherit directly or indirectly from `Exception` but not `RuntimeException` are checked exceptions. These are typically caused by conditions not controlled by the program. For example, in file processing (Chapter 11), if a Java program attempts to open a file that does not exist, a checked `FileNotFoundException` occurs.

The Compiler and Checked Exceptions

The compiler enforces a **`catch`-or-declare requirement** for checked exceptions. If a method's code might throw a checked exception, the method must either

- `catch` the exception by wrapping the code in a `try` block followed by a `catch` handler that deals with the checked-exception type (or one of its superclasses) or
- declare the exception to the method's callers by placing the exception type in a `throws` clause between the method's parameter list and body.

If the code does not satisfy these requirements, a compilation error occurs. **This forces you to think about and deal with potential problems, producing more robust code. If an exception can be handled meaningfully in a method, the method should `catch` the exception rather than declare it.**

The Compiler and Unchecked Exceptions

The compiler does not enforce the `catch`-or-declare requirement for unchecked exceptions, which typically represent preventable coding errors. Ideally, you should fix such errors during program development. However, providing appropriate exception handling code for unchecked exceptions can make your programs more robust. **Unchecked exceptions are not required to be listed**

in a method's `throws` clause.

`throws` Clauses in Overridden Methods

If a subclass method overrides a superclass method, it's an error for the subclass method to list more exceptions in its `throws` clause than the superclass method does. However, a subclass's `throws` clause may list a subset of the types in a superclass's `throws` clause.

Generative AI

1 Prompt genAIs, asking why Java's designers chose to distinguish checked exceptions from unchecked exceptions.
2 Prompt genAIs for a list of common Java unchecked exceptions and how to avoid each.

10.6 `finally` Block

Programs often accomplish their tasks by using system resources, such as memory, files, network connections and database connections. When a program no longer needs a resource, the program should immediately return it to the system to avoid a **resource leak**, which can prevent other programs from using it.

In programming languages such as C and C++, the most common resource leak is a memory leak. Java's automatic garbage collection prevents most memory leaks, but Java will not garbage-collect an object until there are no remaining references to it. So, memory leaks can still occur if a program erroneously keeps references to unneeded objects. Other resources must be explicitly released to ensure they're available for future use.

`finally` Block Placement

A `try` statement's **`finally` block** (also called the **`finally` clause**) consists of the `finally` keyword, followed by code enclosed in curly braces:

- If a `try` statement has one or more `catch` blocks, the `finally` block is **optional** and is placed after the last `catch` block.

- If a `try` statement has no `catch` blocks, the `finally` block is **required** and immediately follows the `try` block.

When the `finally` Block Executes

If present, a `finally` block always executes when its corresponding `try` block terminates, regardless of whether an exception occurs. The only case in which a `finally` block does not execute is if the `try` block calls the **`System` class's `exit` method**, which immediate terminates the program. If an exception occurs but is not caught by the `catch` handlers or if a `catch` block throws an exception, control proceeds to the `finally` block before the program passes the exception to the next enclosing `try` block, typically in a calling method. Because a `finally` block always executes, it's an ideal location for resource-release code that returns files, network connections, database connections and other resources to the system.

Demonstrating the `finally` Block

Figure 10.3 mechanically demonstrates that the `finally` block executes whether or not an exception is thrown in the corresponding `try` block. The program contains `static` methods `main` (lines 5–14), `throwException` (lines 17–35) and `doesNotThrowException` (lines 38–50). Methods `throwException` and `doesNotThrowException` are declared `static`, so `main` can call them directly.

Click here to view code image

```
1  // Fig. 10.3: UsingExceptions.java
2  // try...catch...finally exception handling mechanism.
3
4  public class UsingExceptions {
5     public static void main(String[] args) {
6        try {
7           throwException();
8        }
9        catch (Exception exception) { // exception thrown by throwException
10          System.err.println("Exception handled in main");
11       }
12
13       doesNotThrowException();
14    }
15
16    // demonstrate try...catch...finally
```

```
17      public static void throwException() throws Exception {
18         try { // throw an exception and immediately catch it
19            System.out.println("Method throwException");
20            throw new Exception(); // generate exception
21         }
22         catch (Exception exception) { // catch exception thrown in try
23            System.err.println(
24               "Exception handled in method throwException");
25            throw exception; // rethrow for further processing
26
27            // code here would not be reached; would cause compilation errors
28
29         }
30         finally { // executes regardless of what occurs in try...catch
31            System.err.println("Finally executed in throwException");
32         }
33
34         // code here would not be reached; would cause compilation errors
35      }
36
37      // demonstrate finally when no exception occurs
38      public static void doesNotThrowException() {
39         try { // try block does not throw an exception
40            System.out.println("Method doesNotThrowException");
41         }
42         catch (Exception exception) { // does not execute
43            System.err.println(exception);
44         }
45         finally { // executes regardless of what occurs in try...catch
46            System.err.println("Finally executed in doesNotThrowException");
47         }
48
49         System.out.println("End of method doesNotThrowException");
50      }
51   }
```

```
Method throwException
Exception handled in method throwException
Finally executed in throwException
Exception handled in main
Method doesNotThrowException
Finally executed in doesNotThrowException
End of method doesNotThrowException
```

Fig. 10.3 | `try...catch...finally` exception-handling mechanism.

Throwing Exceptions Using the `throw` Statement

Method `main` (Fig. 10.3) begins executing, enters its `try` block and

line 7 immediately calls method `throwException` (lines 17–35). The `throw` statement in line 20 throws an `Exception` to immediately terminate the `try` block. The object in a `throw` statement must be of a class that directly or indirectly inherits from `Throwable`.

Rethrowing Exceptions

The `Exception` from line 20 is caught at line 22. Then, line 25 **rethrows the exception**. Exceptions typically are rethrown when a `catch` block, upon receiving an exception, can only partially process it. Rethrowing an exception defers the exception handling (or perhaps a portion of it) to another `catch` block associated with an enclosing `try` statement. You rethrow an exception by using `throw`, followed by the exception parameter name, which refers to the exception object that was just caught.

When a rethrow occurs, the next enclosing `try` block gets an opportunity to detect and handle the exception. In this case, the next enclosing `try` block is in lines 6–8 in method `main`. However, before the rethrown exception is returned to line 7 in `main`, the `finally` block in lines 30–32 executes. Then, method `throwException` terminates, `main` detects the rethrown exception in the `try` block and handles it in the `catch` block (lines 9–11).

`finally` Also Executes When No Exceptions Occur

Next, line 13 in `main` calls method `doesNotThrowException` (lines 38–50). No exception is thrown in `doesNotThrowException`'s `try` block (lines 39–41), so the program skips the `catch` block (lines 42–44), but the `finally` block (lines 45–47) nevertheless executes. Then, control proceeds to the statement after the `finally` block (line 49), the function returns to `main`, and the program terminates.

Do Not Throw Exceptions from `finally` Blocks

Do not write code in a `finally` block that can `throw` an exception. If such code is required, enclose it in a `try`…`catch` within the `finally` block to prevent the `finally` from throwing the exception. When a `finally` block throws a new exception while one is still being processed, the first exception is lost.

Do Not Overuse `try` Statements

The `try` statement removes error-processing code from a program's logic, improving program clarity. Do not wrap each statement that

might throw in a separate `try` statement, which decreases readability. Instead, place significant portions of your code in a `try` block, followed by the necessary `catch` blocks and, if needed, a `finally` block (if one is required).

Generative AI

1 Prompt genAIs, asking them which kinds of resource leaks can occur in Java.
2 Prompt genAIs, asking them to explain why you might want to rethrow exceptions.

10.7 Stack Unwinding and Obtaining Information from an Exception

When an exception is thrown but not caught in a particular method, the method-call stack is "unwound" and an attempt is made to `catch` the exception in the next enclosing `try` block. This process is called **stack unwinding**. Unwinding the method-call stack means that the method in which the exception was not caught terminates, all local variables in that method go out of scope and control returns to the statement that invoked that method. If a `try` block encloses that statement, an attempt is made to `catch` the exception. If a `try` block does not enclose that statement or the exception is not caught, stack unwinding occurs again. Figure 10.4 demonstrates stack unwinding, and the exception handler in `main` shows how to access the data in an exception object.[4]

4. If you execute this example using the command `java UsingExceptions.java`—which compiles the program in memory then runs it—this program's output will differ from that of Fig. 10.4.

Click here to view code image

```
1  // Fig. 10.4: UsingExceptions.java
2  // Stack unwinding and obtaining data from an exception object.
3
4  public class UsingExceptions {
5     public static void main(String[] args) {
6        try {
7           method1();
```

```
8          }
9          catch (Exception exception) { // catch exception thrown in method1
10            System.err.printf("%s%n%n", exception.getMessage());
11            exception.printStackTrace();
12
13            // obtain the stack-trace information
14            StackTraceElement[] traceElements = exception.getStackTrace();
15
16            System.out.printf("%nStack trace from getStackTrace:%n");
17            System.out.println("Class\t\tFile\t\t\tLine\tMethod");
18
19            // loop through traceElements to get exception description
20            for (StackTraceElement element : traceElements) {
21               System.out.printf("%s\t", element.getClassName());
22               System.out.printf("%s\t", element.getFileName());
23               System.out.printf("%s\t", element.getLineNumber());
24               System.out.printf("%s%n", element.getMethodName());
25            }
26         }
27      }
28
29      // call method2; throw exceptions back to main
30      public static void method1() throws Exception {
31         method2();
32      }
33
34      // call method3; throw exceptions back to method1
35      public static void method2() throws Exception {
36         method3();
37      }
38
39      // throw Exception back to method2
40      public static void method3() throws Exception {
41         throw new Exception("Exception thrown in method3");
42      }
43   }
```

```
Exception thrown in method3

java.lang.Exception: Exception thrown in method3
        at UsingExceptions.method3(UsingExceptions.java:41)
        at UsingExceptions.method2(UsingExceptions.java:36)
        at UsingExceptions.method1(UsingExceptions.java:31)
        at UsingExceptions.main(UsingExceptions.java:7)

Stack trace from getStackTrace:
Class            File                     Line    Method
UsingExceptions UsingExceptions.java     41      method3
UsingExceptions UsingExceptions.java     36      method2
UsingExceptions UsingExceptions.java     31      method1
```

```
UsingExceptions UsingExceptions.java    7       main
```

Fig. 10.4 | Stack unwinding and obtaining data from an exception object.

Stack Unwinding

In `main`, the `try` block (lines 6–8) calls `method1` (declared at lines 30–32), which in turn calls `method2` (declared at lines 35–37), which in turn calls `method3` (declared at lines 40–42). The program then proceeds as follows:

- Line 41 in `method3` throws an `Exception` object—this is the throw point.
- Because the `throw` statement is not enclosed in a `try` block, stack unwinding occurs. So, `method3` terminates at line 41, then returns control to line 36—the statement in `method2` that invoked `method3`.
- Because no `try` block encloses line 36, stack unwinding occurs again. So, `method2` terminates at line 36 and returns control to line 31—the statement in `method1` that invoked `method2`.
- Because no `try` block encloses line 31, stack unwinding occurs again. So, `method1` terminates at line 31 and returns control to line 7—the statement in `main` that invoked `method1`.
- The `try` block at lines 6–8 encloses this statement. The exception has not been handled, so the `try` block terminates, and the first matching `catch` block (lines 9–26) catches and processes the exception.

If there were no matching `catch` blocks, and the checked exception is not declared in each method that throws it, a compilation error would occur—`main` catches the exception so it does not have a `throws` clause.

Obtaining Data from an Exception Object

All exceptions derive from `Throwable`, which has a **`printStackTrace` method** that outputs the stack trace to the standard error stream. Often, this is helpful in testing and debugging. Class `Throwable` also provides a **`getStackTrace` method**

that retrieves the stack-trace information that might be printed by `printStackTrace`. Class `Throwable`'s **`getMessage` method** (inherited by all `Throwable` subclasses) returns the descriptive string stored in an exception. `Throwable` method `toString` (also inherited by all `Throwable` subclasses) returns a `String` containing the name of the exception's class and a descriptive message.

Java's default exception handler runs if an exception is not caught, displaying the name of the exception, a descriptive message that indicates the problem that occurred and a complete execution stack trace. In an application with a single thread of execution, the application terminates. In an application with multiple threads, the thread that caused the exception terminates. We discuss multithreading in Chapter 18.

The `catch` handler in Fig. 10.4 (lines 9–26) demonstrates `getMessage`, `printStack-Trace` and `getStackTrace`. If we wanted to output the stack-trace information to streams other than the standard error stream, we could use the information returned from `getStackTrace` and output it to another stream or use one of the overloaded versions of `printStackTrace`. Sending data to other streams is discussed in Chapter 11.

Line 10 invokes the exception's `getMessage` method to get the exception description. Line 11 invokes the exception's `printStackTrace` method to output the stack trace that indicates where the exception occurred. Line 14 invokes the exception's `getStackTrace` method to obtain the stack-trace information as an array of **`StackTraceElement`** objects for more detailed inspection. Lines 20–25 get each `StackTraceElement` in the array and invoke its **methods `getClassName`**, **`getFileName`**, **`getLineNumber`** and **`getMethodName`** to get the class name, filename, line number and method name, respectively, for that `Stack-TraceElement`. Each `StackTraceElement` represents one method call on the method-call stack.

The program's output shows that the information `printStackTrace` displays follows the pattern:

Click here to view code image

className.methodName(*fileName*:*lineNumber*)

where *className*, *methodName* and *fileName* are the class, method and file in which the exception occurred, and *lineNumber* is where in the file the exception occurred. Method `getStackTrace` enables custom processing of the exception information. Compare the

output of `printStackTrace` with the output created from the `StackTraceElement`s to confirm that both contain the same stack-trace information.

Stack-Walking API

`Throwable` methods `printStackTrace` and `getStackTrace` each process the entire method-call stack. When debugging, this can be inefficient. For example, you may be interested only in stack frames corresponding to methods of a specific class. Java's **Stack-Walking API** (class **`StackWalker`** in package `java.lang`) uses lambdas and streams (Chapter 14) to access method-call-stack information more efficiently. You can learn more about this API at:

```
https://openjdk.org/jeps/259
```

Generative AI

1 Prompt genAIs, asking for real-world use-cases of `StackTraceElement`s.

10.8 Chained Exceptions

Sometimes, a method responds to an exception by throwing a different exception type specific to the current application. If a `catch` block throws a new exception, the original exception's information and stack trace are lost. Earlier Java versions provided no mechanism to wrap the original exception information with the new exception's information to provide a complete stack trace showing where the original problem occurred, making debugging more difficult. **Chained exceptions** enable an exception object to maintain the complete stack-trace information from the original exception. Figure 10.5 demonstrates chained exceptions.

Click here to view code image

```
1  // Fig. 10.5: UsingChainedExceptions.java
2  // Chained exceptions.
3
4  public class UsingChainedExceptions {
```

```
5       public static void main(String[] args) {
6          try {
7             method1();
8          }
9          catch (Exception exception) { // exceptions thrown from method1
10            exception.printStackTrace();
11         }
12      }
13
14      // call method2; throw exceptions back to main
15      public static void method1() throws Exception {
16         try {
17            method2();
18         }
19         catch (Exception exception) { // exception thrown from method2
20            throw new Exception("Exception thrown in method1", exception);
21         }
22      }
23
24      // call method3; throw exceptions back to method1
25      public static void method2() throws Exception {
26         try {
27            method3();
28         }
29         catch (Exception exception) { // exception thrown from method3
30            throw new Exception("Exception thrown in method2", exception);
31         }
32      }
33
34      // throw Exception back to method2
35      public static void method3() throws Exception {
36         throw new Exception("Exception thrown in method3");
37      }
38   }
```

```
java.lang.Exception: Exception thrown in method1
        at UsingChainedExceptions.method1(UsingChainedExceptions.java:20)
        at UsingChainedExceptions.main(UsingChainedExceptions.java:7)
Caused by: java.lang.Exception: Exception thrown in method2
        at UsingChainedExceptions.method2(UsingChainedExceptions.java:30)
        at UsingChainedExceptions.method1(UsingChainedExceptions.java:17)
        ... 1 more
Caused by: java.lang.Exception: Exception thrown in method3
        at UsingChainedExceptions.method3(UsingChainedExceptions.java:36)
        at UsingChainedExceptions.method2(UsingChainedExceptions.java:27)
        ... 2 more
```

Fig. 10.5 | Chained exceptions.

Program Flow of Control

The program has four methods—`main` (lines 5–12), `method1` (lines 15–22), `method2` (lines 25–32) and `method3` (lines 35–37)—and operates as follows:

- Line 7 in `main`'s `try` block calls `method1`. Line 17 in `method1`'s `try` block calls `method2`. Line 27 in `method2`'s `try` block calls `method3`.
- In `method3`, line 36 throws a new `Exception`. Because line 36 is not in a `try` block, `method3` terminates, and the exception is returned to line 27 in the calling method (`method2`).
- Line 27 is in a `try` block, so the `try` block terminates, and the exception is caught at lines 29–31.
- Line 30 in the `catch` block throws a new exception. We call the `Exception` constructor with two arguments, where the second represents the exception that was the original cause of the problem—every exception class has a constructor like this. In this program, the original exception occurred at line 36. Because an exception is thrown from the `catch` block, `method2` terminates and returns the new exception to line 17 in `method1`.
- Line 17 is in a `try` block, so the `try` block terminates, and the exception is caught at lines 19–21.
- Line 20 in the `catch` block throws a new exception, using the exception that was caught as the constructor's second argument. Because an exception is thrown from the `catch` block, `method1` terminates and returns the new exception to line 7 in `main`.
- The `try` block in `main` terminates, and the exception is caught at lines 9–11. Line 10 prints a stack trace containing all the information from the chained exceptions.

`Throwable` Method `getCause`

For any chained exception, you can get the `Throwable` that initially caused that exception by calling **`Throwable` method `getCause`**.

Program Output

In the program output:

- the first three lines show the most recent exception thrown (i.e., the one from `method1` at line 20),
- the next four lines indicate the exception thrown from `method2` at line 30, and
- the last four lines represent the exception thrown from `method3` at line 36.

As you read the output from bottom to top, it shows how many more chained exceptions remain.

Generative AI

1 Prompt genAIs, asking for a real-world example of "exception translation" in which a Java program would catch an exception, then use chained exceptions to add context before throwing a more specific exception.

10.9 Declaring Custom Exceptions

Most Java programmers use existing classes from the Java API, third-party vendors and freely available class libraries from sites like GitHub to build Java applications. The methods of those classes may throw appropriate exceptions when problems occur. You write code that processes these existing exceptions to make your programs more robust. If you build classes that other programmers will use, it's often appropriate to declare your own exception classes specific to the problems that can occur when another programmer uses your classes. Associating each type of serious execution-time malfunction with an appropriately named `Exception` subclass improves program clarity.

Extending an Existing Exception Class

A new exception class must extend an existing one to ensure the new class can be used with the exception-handling mechanism. The new class's name should end with "`Exception`" by convention. An exception class typically contains four constructors:

- one that takes no arguments and passes a default error

message `String` to the superclass constructor,

- one that receives a customized error message as a `String` and passes it to the superclass constructor,
- one that receives a customized error message as a `String` and a `Throwable` (for chaining exceptions) and passes both to the superclass constructor, and
- one that receives a `Throwable` (for chaining exceptions) and passes it to the super-class constructor.

Use an Existing Exception Class if Possible

Most programmers will not need to declare their own exception classes. Before defining your own, study the Java API's existing ones. If there is no appropriate existing class, try to extend a related exception class. For example, if you're creating a new class to represent when a method attempts a division by zero, you might extend class `ArithmeticException` because division by zero occurs during arithmetic.

Choosing Whether Custom Exceptions Should be Checked or Unchecked

If the existing classes are not appropriate superclasses for your new exception class, decide whether your new class should be a checked or an unchecked exception class:

- If clients should be required to handle the exception, the new exception class should be a checked exception that extends `Exception` but not `RuntimeException`. In this case, the client application should be able to reasonably recover from the exception.
- If the client code can ignore the exception (i.e., the exception is an unchecked one), the new exception class should extend `RuntimeException`.

Generative AI

1 Prompt genAIs, asking for guidelines on when to create new exception types in Java.
2 Prompt genAIs, asking for guidelines on when to make a custom exception a checked vs. unchecked exception.

10.10 Preconditions and Postconditions

Programmers spend significant amounts of time maintaining and debugging code. To facilitate these tasks and to improve the overall design, you can specify the expected states before and after a method's execution. These states are called preconditions and postconditions, respectively.

Preconditions

A **precondition** must be true when a method is invoked. Preconditions describe constraints on method parameters and any other expectations the method has about the current state of a program just before it begins executing. **If the preconditions are not met, then the method's behavior is undefined**—it may throw an exception, proceed with an illegal value or attempt to recover from the error. You should not expect consistent behavior if the preconditions are not satisfied.

Postconditions

A **postcondition** is true after the method successfully returns. Postconditions describe constraints on the return value and any other side effects the method may have. **When defining a method, you should document all postconditions so that others know what to expect when they call your method, and you should ensure that it honors all its postconditions if its preconditions are met.**

Throwing Exceptions When Preconditions or Postconditions Are Not Met

When preconditions or postconditions are not met, methods typically throw exceptions. For example, examine `String` method `charAt`, which has one `int` parameter—an index in the `String`. Method `charAt` requires as a precondition that the index it receives is greater than or equal to zero and less than the `String`'s length. If the precondition is met, the postcondition states that the method will return the character at the index specified by its argument. Otherwise, `charAt` throws an `IndexOutOfBoundsException`. We trust that method `charAt` satisfies its postcondition, provided that we

meet the precondition. We need not be concerned with the details of how the method actually retrieves the character at the index.

Documenting Preconditions or Postconditions

Typically, a method's preconditions and postconditions are described as part of its specification. When designing your own methods, you typically state the preconditions and postconditions in a comment before the method declaration.

Generative AI

1 Prompt genAIs to list typical kinds of preconditions and postconditions for Java methods.
2 Prompt genAIs, asking which exceptions a Java method typically throws if a precondition or postcondition is not met.
3 Prompt genAIs, asking for a simple Java program demonstrating a method with properly commented preconditions and postconditions, which are tested via `if` statements that `throw` exceptions if a precondition or postcondition is not met.

10.11 Assertions

As you implement and debug a class, it's sometimes helpful to state conditions that should be true at a particular point in a method. These conditions, called **assertions**, help ensure a program's validity by catching potential bugs and identifying possible logic errors during development. Preconditions and postconditions are two types of assertions. Preconditions are assertions about a program's state when a method is invoked, and postconditions are assertions about its state after a method finishes.

The `assert` Statement

While assertions can be stated as comments to guide you during program development, Java includes two versions of the **`assert` statement** to validate assertions programmatically. The `assert` statement evaluates a `boolean` expression and, if `false`, throws an **`AssertionError`** (a subclass of `Error`). The first form of the `assert` statement is

```
assert expression;
```

which throws an `AssertionError` if *expression* is `false`. The second form is

Click here to view code image

```
assert expression1 : expression2;
```

which evaluates *expression1* and throws an `AssertionError` with *expression2* as the error message if *expression1* is `false`.

Using `assert`

During development, you can use assertions to implement preconditions and postconditions programmatically or verify any other intermediate states that help ensure your code is working correctly.

Assertions are primarily for debugging and identifying logic errors in an application. **They are ignored by default, so you must explicitly enable them when executing a program** as follows—the `-ea` option stands for "enable assertions":

```
java -ea AssertTest
```

Figure 10.6 demonstrates the `assert` statement. Line 9 prompts the user to enter a number between 0 and 10, then line 10 reads the number. Line 13 determines whether the user entered a number within the valid range. If the number is out of range, the `assert` statement reports an error; otherwise, the program proceeds normally. **You should not use `assert` to indicate runtime problems in production code**, as we do in Fig. 10.6 only for demonstration purposes. Production code should throw exceptions to indicate problems encountered during execution.

Click here to view code image

```
1  // Fig. 10.6: AssertTest.java
2  // Checking with assert that a value is within range
3  import java.util.Scanner;
4
5  public class AssertTest {
6     public static void main(String[] args) {
7        var input = new Scanner(System.in);
8
9        System.out.print("Enter a number between 0 and 10: ");
```

```
10          int number = input.nextInt();
11
12          // assert that the value is >= 0 and <= 10
13          assert (number >= 0 && number <= 10) : "bad number: " + number;
14
15          System.out.printf("You entered %d%n", number);
16       }
17    }
```

```
Enter a number between 0 and 10: 5
You entered 5
```

```
Enter a number between 0 and 10: 50
Exception in thread "main" java.lang.AssertionError: bad number: 50
        at AssertTest.main(AssertTest.java:13)
```

Fig. 10.6 | Checking with `assert` that a value is within range.

Do Not Catch `AssertionError`s

`AssertionError`s should be used only during program development to identify problems in your code. Do not catch them. Simply allow your program to terminate so you can then locate and fix the source of the problem to make your program more robust.

Generative AI

1 Prompt genAIs, asking for a simple Java program demonstrating a method with properly commented preconditions and postconditions that are tested via `assert`.

10.12 `try`-with-Resources Statement: Automatic Resource Deallocation

In early Java versions, you'd place resource-release code in a `finally` block to ensure that a resource was released, regardless of whether the corresponding `try` block that used the resource threw exceptions. The **`try`-with-resources statement** simplifies writing code in which you obtain one or more resources, use them in a `try`

block and release them in a corresponding `finally` block. Most Java API classes representing resources that need to be released now implement the **`AutoCloseable` interface** and provide a `close` method. Objects of such classes may be used in a `try`-with-resources statement as follows:

Click here to view code image

```
try (var theObject = new ResourceClass()) {
   // Use theObject here. Resources automatically released when the
   // try block terminates successfully or due to an exception.
}
catch (Exception e) {
   // Catch exceptions that occur while using the resource.
}
```

where the *ResourceClass* in parentheses must implement `AutoCloseable`. The `try`-with-resources statement is the preferred way to manage `AutoCloseable` resources. The `try` block uses the *ResourceClass* object, then calls its `close` method to release the its resources when the block terminates successfully or due to an exception. The `try`-with-resources `catch` clause is optional. You can create multiple `AutoCloseable` objects in the parentheses following `try` by separating them with a semicolon (`;`). We'll use the `try`-with-resources statement in several programs starting in Chapter 11, Files, I/O Streams, JSON Serialization & CSV Files.

`try`-with-Resources Can Use Effectively `final` Variables

If the compiler can infer that a local variable could have been declared `final` because the enclosing method never modifies it after it's declared and initialized, then it's **effectively `final`**. You can create an `AutoCloseable` object and assign it to a local variable that's explicitly declared `final` or that's effectively `final`. Then, you can use the variable in a `try`-with-resources statement that automatically releases the object's resources:

Click here to view code image

```
var theObject = new ResourceClass();

try (theObject) {
   // Use theObject here. Resources automatically released when the
   // try block terminates successfully or due to an exception.
}
catch (Exception e) {
```

```
        // Catch exceptions that occur while using the resource.
    }
```

As before, you can separate with a semicolon (`;`) multiple `AutoCloseable` objects in the parentheses following `try`. This simplifies the `try`-with-resources statement's code, especially for cases in which the statement uses and releases multiple `AutoCloseable` objects.

Suppressed Exceptions

When an exception is thrown in a `try`-with-resources statement's `try` block, the statement immediately closes the `AutoCloseable` resources specified in the `try`'s parentheses. If any exceptions occur while closing those resources, Java suppresses the new ones so the original exception can continue being processed. The **suppressed exceptions** are added to the original exception object. You can access the suppressed exception(s) via an exception object's **`getSurpressed` method** (inherited from `Throwable` by all exception classes), which returns an array of the suppressed `Throwable` objects.

Generative AI

1 Prompt genAIs, asking them to list commonly used Java API classes that implement the `AutoCloseable` interface and mention the resource each manages.
2 Prompt genAIs, asking them for a typical example of a suppressed exception in Java's `try`-with-resources statement.

10.13 Unnamed Variables in `catch` Handlers

Java 22 introduced **unnamed variables (_)**[5] for cases where a variable name is required but you do not need to use the variable. For example, Fig. 10.7 creates an array, then attempts to access an out-of-bounds element, which throws an `ArrayIndexOutOfBoundsException`. A `catch` handler must specify a variable name after the exception type. However, if your `catch`

handler does not require access to the exception object, you can use the unnamed variable _ in place of a variable name (line 10).

5. "JEP 456: Unnamed Variables & Patterns." Accessed September 22, 2024. https://openjdk.org/jeps/456.

Click here to view code image

```
1  // Fig. 10.7: UnnamedVariableDemo.java
2  // Using _ where a variable name is required.
3  public class UnnamedVariableDemo {
4     public static void main(String[] args) {
5        int[] array = {1, 2, 3, 4, 5};
6
7        try {
8           int value = array[10]; // index 10 is out of bounds
9        }
10       catch (ArrayIndexOutOfBoundsException _) {
11          System.out.println("Index out of range");
12       }
13    }
14 }
```

```
Index out of range
```

Fig. 10.7 | Using _ where a variable name is required.

Unnamed Variables in `record` Patterns

You may also use unnamed variables in `record` patterns that decompose matching `record` objects in

- pattern-matching `instanceof` expressions (Section 9.8.4) and
- pattern-matching `switch` expressions (Section 8.20.1)

If a match requires only a subset of the `record`'s elements, the `record` pattern can use _ in place of the items you wish to ignore. For the complete list of other _ uses, visit

```
https://openjdk.org/jeps/456
```

10.14 Wrap-Up

In this chapter, you used exception handling via the `try` statement to deal with execution-time errors. You learned that exception

handling enables you to remove error-handling code from the "main line" of the program's execution, making your code easier to read. We showed how to use `try` blocks to enclose code that might `throw` exceptions and how to use `catch` blocks to deal with exceptions that arise. We also introduced multi-`catch` for processing several exceptions in the same `catch` handler.

You learned about the termination model of exception handling, in which program control does not return to the `throw` point after an exception is handled. We discussed checked vs. unchecked exceptions and how to specify with the `throws` clause the exceptions that a method might `throw`.

We showed that the `finally` block always executes, regardless of whether an exception occurs in the corresponding `try` block, making the `finally` block good for resource deallocation. You also learned how to re-`throw` exceptions. We showed how to obtain information about an exception using methods `printStackTrace`, `getStackTrace` and `getMessage`. We presented chained exceptions, which allow you to throw a more specific exception while maintaining the original exception and its stack trace information. Then, we showed how to create your own exception classes.

We introduced preconditions and postconditions to help programmers using your methods understand conditions that must be true when the method is called and when it returns, respectively. When preconditions and postconditions are not met, methods typically `throw` exceptions. We discussed the `assert` statement and how it can be used to help you debug your programs.

We introduced and the `try`-with-resources statement for automatically deallocating an `AutoCloseable` resource after it's used in the `try` block. Finally, we demonstrated an unnamed variable (_) in a `catch` handler that did not need access to the exception it caught. In the next chapter, you'll learn about file processing, including how persistent data is stored and retrieved.

Special Section: Generative AI Exercises

For each of the following questions, try multiple genAIs to determine which performs best. Always verify the results. Keep in mind that they sometimes make mistakes and even hallucinate, creating unreliable information.

10.1 ***(Generative AI: finally Blocks)*** Prompt genAIs with the following: Most classes that manage resources in Java implement

the `AutoCloseable` interface so they can be used with the `try`-with-resources statement. Given that, provide a real-world example in which a `try` statement's `finally` block should be used.

10.2 ***(Generative AI: Postconditions)*** Prompt genAIs, asking for a list of postconditions used in the `java.lang.Math` class's methods.

10.3 ***(Generative AI: Advantages and Disadvantages of Custom Exceptions)*** Prompt genAIs, asking for the advantages and disadvantages of creating custom exceptions in Java.

10.4 ***(Generative AI: Do Organizations Discourage Exceptions)*** Prompt genAIs, asking whether some Java software-development organizations discourage the use of exceptions. If so, why?

10.5 ***(Generative AI: Exception-Handling Performance)*** Prompt genAIs, asking for a list of performance considerations when using Java exception handling.

10.6 ***(Generative AI: Exceptions Thrown By BigDecimal Methods)*** Prompt genAIs, asking for a list of `java.math.BigDecimal` methods that throw exceptions and why those exceptions might occur.

10.7 ***(Generative AI: Logging Exceptions)*** Prompt genAIs, asking what exception logging is and what the pros and cons of using it are.

10.8 ***(Generative AI: Swallowing Exceptions)*** Prompt genAIs, asking what it means to "swallow an exception" and why you should not do it.

10.9 ***(Generative AI: Avoiding Catastrophic Issues)*** Prompt genAIs, asking how exception handling can be used to avoid catastrophic issues.

10.10 ***(Generative AI: Using RuntimeExceptions to Prevent "throws Chains")*** Prompt genAIs, asking why it's popular in industry to use Java `RuntimeException`s "everywhere" to prevent "`throws` chains" all the way up the stack.

0 1 2 3 4 5 6 7 8 9 . . . n-1
. . .
end-of-file marker

Files, I/O Streams, JSON Serialization & CSV Files

Objectives

In this chapter, you'll:

- Retrieve information about files and directories.
- Create, read, write and update files.
- Learn the differences between text files and binary files.
- Use class `Formatter` to output text to a file.
- Use class `Scanner` to input text from a file.
- Write objects to and read objects from a file using JavaScript Object Notation (JSON) serialization and deserialization with the Jackson open-source library.
- Get a city's weather report in JSON format by invoking a free OpenWeatherMap web service.
- Write to and read from comma-separated-value (CSV) files.
- Explore an introductory data analytics case study that reads and processes *Titanic* disaster data from a CSV file.
- Explore an objects-natural case study on RSA public-key cryptography, which is used with AES (Chapter 7) to secure data stored in files and transmitted over networks.
- Continue using generative AI to leverage your Java learning experience.

Outline

11.1 Introduction

11.2 Files and Streams

11.3 Using NIO Classes and Interfaces to Get File and Directory Information

11.4 Sequential Text Files

11.4.1 Creating a Sequential Text File

11.4.2 Reading Data from a Sequential Text File
11.4.3 Updating Sequential Files
11.5 Case Study: JavaScript Object Notation (JSON) Serialization
11.5.1 Serializing a `List<Account>`
11.5.2 Deserializing a `List<Account>`
11.6 Case Study: Processing a JSON Response from a Web Service
11.7 Case Study: Creating and Reading CSV Files
11.7.1 Creating a CSV File
11.7.2 Reading a CSV File
11.8 Case Study: Reading and Analyzing a CSV File Containing the Titanic Disaster Dataset
11.9 Objects-Natural Security Case Study: RSA Public-Key Cryptography
11.10 Wrap-Up
Special Section: Generative AI Exercises

11.1 Introduction

Data in memory is temporary. Files are used for **data persistence**—permanent data retention for data that should remain available after a program terminates. Computers store files on **secondary storage devices**, such as flash drives, and frequently today, in the cloud. This chapter explains how Java programs create and process files containing human-and-computer-readable data using some of Java's extensive file-processing APIs and a popular free, open-source library. Another way to persist data is by using databases, which we discuss in Chapter 20, Accessing Databases with JDBC and SQLite.

We begin with Java's architecture for handling files programmatically. Next, we explain that data can be stored in text or binary files, and discuss their differences. We demonstrate using NIO (short for new input/output) classes and interfaces to retrieve information about files and directories. Then, we use `Formatter` objects to write to text files and `Scanner` objects to read from them.

We create and manipulate sequential text files and present several case studies that demonstrate transforming objects to and from popular data formats. For simplicity, several examples use type `double` for monetary amounts—recall that real applications should use `BigDecimal` for monetary amounts.

JavaScript Object Notation (JSON) and Invoking a Web Service

The first case study introduces JavaScript Object Notation (JSON)—a human-and-computer-readable text format. We use the popular Jackson open-source library to write objects' JSON representations to a file (a process known as serialization), then read the JSON and recreate the objects (a process known as deserialization). In the second JSON case study, we use networking features from the Java API's `HttpClient`, `HttpRequest` and `HttpResponse` classes (package `java.net.http`) to invoke an `OpenWeatherMap.org` web service that returns a city's weather report in JSON format, then use Jackson to process the response and display the weather report.

Comma-Separated-Values (CSV) Files and Basic Data Analytics

The next two case studies introduce the CSV (comma separated values) file format and how to write and read CSV files using the Jackson open-source library. CSV is popular for datasets used in big data, data analytics, and data science, as well as artificial intelligence applications like natural language processing, machine learning, and deep learning.

A commonly used dataset for data-analytics beginners is the Titanic disaster dataset, which lists the passengers and whether they survived when the ship struck an iceberg and sank during its first voyage on April 10–15, 1912. This case study uses the Jackson library to read the dataset from a CSV file. Then, we view some of the data and perform basic data analytics.

Objects-Natural Security Case Study: RSA Public-Key Cryptography

We presented objects-natural case studies in Chapters 2–7 in which you conveniently worked with objects of existing Java API classes before you learned to develop custom classes in Chapter 8. In our final objects-natural case study, we continue emphasizing security

and privacy by enhancing our private-key cryptography example from Section 7.9. We use RSA public-key cryptography to encrypt and decrypt the AES secret key. This enables secure transmission to an encrypted message's recipient. Such encryption and decryption techniques are critical for securing data stored in files and transmitted over networks.

11.2 Files and Streams

Java views each file as a sequential **stream of bytes**:[1]

1. Java's NIO APIs also include classes and interfaces that implement so-called channel-based architecture for high-performance I/O. These topics are beyond the scope of this book.

Every operating system provides a mechanism to determine the end of a file, such as an **end-of-file marker** or a count of the total bytes in the file that's recorded in a system-maintained administrative data structure. A Java program processing a stream of bytes receives an indication from the operating system when it reaches the end of the stream—the program does not need to know how the underlying platform represents files or streams. In some cases, the end-of-file indication occurs as an exception. In others, the indication is a return value from a method invoked on a stream-processing object.

Byte-Based and Character-Based Streams

File streams can be used to input and output data as bytes or characters:

- **Byte-based streams** output and input data in its binary format—an `int` is four bytes, a `double` is eight bytes, etc. Such streams are used for non-text data, such as images, videos or audio files. We'll process binary data in **Chapter 19, Building API-Based Java Generative AI Applications**.
- **Character-based streams** output and input text data as sequences of characters. Every character is two bytes, so the number of bytes for a given value depends on its number of characters. For example, the value `2000000000` (10 characters) requires 20 bytes, but the value `7` (one character)

requires only two bytes.

Files manipulated with byte-based streams are **binary files**, while those manipulated with character-based streams are **text files**. Text files can be read by text editors, while binary files are read by programs that understand the file's specific content. You can use a numeric value (e.g., `5`) from a binary file directly in calculations. In contrast, the character `'5'` in a text file is just part of a `String`, like in the sentence `"Sarah is 15 years old."`

Standard Input, Standard Output and Standard Error Streams

A Java program **opens a file** by creating an object and associating a stream of bytes or characters with it. The object's constructor interacts with the operating system to open the file. Java can also associate streams with different devices. When a Java program begins executing, it creates three stream objects that are associated with devices:

- The `System.in` (standard input stream) object normally enables a program to obtain input from the keyboard.
- The `System.out` (standard output stream) object normally enables a program to output data to the screen.
- The `System.err` (standard error stream) object normally enables a program to output error messages to the screen.

These streams can be **redirected** via the **`System` methods `setIn`, `setOut` and `setErr`**, respectively. For `System.in`, this enables the program to read bytes from a different source. For `System.out` and `System.err`, the output can be sent to a different location, such as a file on your system.

The `java.io` and `java.nio` Packages

Java programs perform stream-based processing with classes and interfaces from package **`java.io`** and the subpackages of **`java.nio`**. Other packages throughout the Java APIs also contain features based on those in the `java.io` and `java.nio` packages.

Character input and output can be performed with classes `Scanner` and **`Formatter`**, as you'll see in Section 11.4. You've used `Scanner` extensively to input data from the keyboard. `Scanner` can also read data from a file. Class `Formatter` enables formatted data

to be output to any text-based stream similar to `System.out.printf`. Appendix B, Formatted Output, presents the details of formatted output with `printf`. All these features can be used to format text files as well.

Another Type of Stream

Chapter 14, Functional Programming with Lambdas & Streams, introduces a type of stream used to process collections of elements (like arrays and `ArrayList`s) rather than the streams of bytes and characters we discuss in this chapter. Section 14.13 shows how to use lambdas and streams with a text file.

Generative AI

1 Prompt genAIs to compare byte-based and character-based Java I/O streams and discuss when you'd use each.
2 Prompt genAIs, asking for real-world examples of Java byte-based and character-based streams.

11.3 Using NIO Classes and Interfaces to Get File and Directory Information

Package `java.nio.file` provides features useful for retrieving information about files and directories:

- **`Path` interface**—Objects of classes that implement `Path` represent the location of a file or directory. `Path` objects do not open files or provide any file-processing capabilities. Among its capabilities, the `Path` interface provides `static` methods to create `Path` objects representing file or directory locations.
- **`Files` class**—This class provides `static` methods for common file and directory manipulations, such as copying files; creating and deleting files and directories; getting information about files and directories; reading the contents of files; getting objects that allow you to manipulate the contents of files and directories; and more.

- **`DirectoryStream` interface**—Objects of classes that implement this interface enable a program to iterate through a directory's contents.

Absolute vs. Relative Paths

A file or directory's path specifies its location on your computer. The path includes some or all of the directories leading to the file or directory:

- An **absolute path** specifies a complete location, starting with the **root directory**, that leads to a specific file or directory. Every file or directory on a particular disk drive or solid-state drive (SSD) has the same root directory in its path.
- A **relative path** is "relative" to a directory—for example, a path relative to the directory in which the program began executing. In relative paths, dot (`.`) represents the current folder, and dot dot (`..`) represents the enclosing folder.

Example: Getting File and Directory Information

Figure 11.1 prompts the user to enter a file or directory name, then uses `Path`, `Files` and `DirectoryStream` to output metadata about that file or directory (also called a folder), including:

- whether the file or folder exists,
- its name,
- whether it's a file or folder,
- whether its location is an absolute path,
- when it was last modified,
- its size,
- its path and
- its absolute path—if the user enters an absolute path, the path and absolute path will be the same.

For a folder, we also list its contents. In the first sample output, we entered the `Path` for the folder containing this chapter's examples. In the second, we entered the `Path` for this example's source-code file. In both cases, we specified absolute paths.

Click here to view code image

```java
// Fig. 11.1: FileAndDirectoryInfo.java
// File class used to obtain file and directory information.
import java.io.IOException;
import java.nio.file.DirectoryStream;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.Scanner;

public class FileAndDirectoryInfo {
   public static void main(String[] args) throws IOException {
      var input = new Scanner(System.in);

      System.out.println("Enter file or directory name:");

      // create Path object based on user input
      Path path = Path.of(input.nextLine());

      if (Files.exists(path)) { // if path exists, output info about it
         // display file or directory information
         System.out.printf("%n%s exists%n", path.getFileName());
         System.out.printf("%s a directory%n",
            Files.isDirectory(path) ? "Is" : "Is not");
         System.out.printf("%s an absolute path%n",
            path.isAbsolute() ? "Is" : "Is not");
         System.out.printf("Last modified: %s%n",
            Files.getLastModifiedTime(path));
         System.out.printf("Size: %s%n", Files.size(path));
         System.out.printf("Path: %s%n", path);
         System.out.printf("Absolute path: %s%n", path.toAbsolutePath());

         if (Files.isDirectory(path)) { // output directory listing
            System.out.printf("%nDirectory contents:%n");

            // object for iterating through a directory's contents
            DirectoryStream<Path> directoryStreamPath =
               Files.newDirectoryStream(path);

            for (Path currentPath : directoryStreamPath) {
               System.out.println(currentPath);
            }
         }
      }
      else { // not file or directory, output error message
         System.out.printf("%s does not exist%n", path);
      }
   }
}
```

```
Enter file or directory name:
c:\Users\pauldeitel\Documents\examples\ch11
```

```
ch11 exists
Is a directory
Is an absolute path
Last modified: 2024-11-29T16:30:04.8490973Z
Size: 4096
Path: C:\Users\pauldeitel\Documents\examples\ch11
Absolute path: C:\Users\pauldeitel\Documents\examples\ch11

Directory contents:
C:\Users\pauldeitel\Documents\examples\ch11\fig11_01
C:\Users\pauldeitel\Documents\examples\ch11\fig11_02
C:\Users\pauldeitel\Documents\examples\ch11\fig11_03
C:\Users\pauldeitel\Documents\examples\ch11\fig11_04
C:\Users\pauldeitel\Documents\examples\ch11\fig11_05
C:\Users\pauldeitel\Documents\examples\ch11\fig11_07
C:\Users\pauldeitel\Documents\examples\ch11\fig11_08
C:\Users\pauldeitel\Documents\examples\ch11\fig11_09
C:\Users\pauldeitel\Documents\examples\ch11\fig11_10
C:\Users\pauldeitel\Documents\examples\ch11\fig11_11
C:\Users\pauldeitel\Documents\examples\ch11\libs
```

```
Enter file or directory name:
c:\Users\pauldeitel\Documents\examples\ch11\fig11_01\FileAndDirectoryInfo.java

FileAndDirectoryInfo.java exists
Is not a directory
Is an absolute path
Last modified: 2024-11-28T20:22:22.8167285Z
Size: 2992
Path: c:\Users\pauldeitel\Documents\examples\ch11\fig11_01\FileAndDirectoryIn-
fo.java
Absolute path: c:\Users\pauldeitel\Documents\examples\ch11\fig11_01\FileAndDi-
rectoryInfo.java
```

Fig. 11.1 | `File` class used to obtain file and directory information.

Line 13 prompts the user for a file or directory.[2] Line 16 inputs the filename or directory name and passes it to **`Path` static method `of`**, which converts the `String` to a `Path`. Line 18 invokes **`Files` static method `exists`**, which receives a `Path` and determines whether it exists (either as a file or as a directory). If the name does not exist, line 44 displays a message containing the `Path`'s `String` representation followed by "`does not exist`." Otherwise, lines 20–41 execute:

2. In Section 16.7, we'll update this example with JavaFX `FileChooser` and `DirectoryChooser` dialogs that enable you to select a file or

directory rather than entering it

- **`Path` method `getFileName`** (line 20) gets the `String` name of the file or directory without any location information.
- **`Files static` method `isDirectory`** (line 22) receives a `Path` and returns a `boolean` indicating whether that `Path` represents a directory.
- **`Path` method `isAbsolute`** (line 24) returns a `boolean` indicating whether that `Path` represents an absolute path to a file or directory.
- **`Files static` method `getLastModifiedTime`** (line 26) receives a `Path` and returns a `FileTime` (package `java.nio.file.attribute`) indicating when the file was last modified. The program outputs the `FileTime`'s default `String` representation.
- **`Files static` method `size`** (line 27) receives a `Path` and returns a `long` representing the number of bytes in the file or directory. For directories, the value returned is platform-specific.
- **`Path` method `toString`** (called implicitly at line 28) returns a `String` representing the `Path`.
- **`Path` method `toAbsolutePath`** (line 29) converts the `Path` on which it's called to an absolute path.

If the `Path` represents a directory (line 31), lines 35–36 use the **`Files` class's `static` method `newDirectoryStream`** to get a `DirectoryStream<Path>` containing `Path` objects for the directory's contents. `DirectoryStream` is a generic type like `ArrayList`. Lines 38–40 display the `String` representation of each `Path` in the `DirectoryStream<Path>`.

Once you've confirmed that a `Path` exists, the methods Fig. 11.1 demonstrates may still throw `IOException`s—we do not `catch` these checked exceptions in this program, so we added a `throws` clause to `main` (line 10). For example, the file or directory represented by the `Path` could be deleted from the system after the call to `Files` method `exists` and before the other statements in lines 20–41 execute. Industrial strength file- and directory-processing programs require extensive exception handling to deal with such possibilities.

Separator Characters

A **separator character** is used to separate directories and files in a path. On a Windows computer, the separator character is a backslash (`\`). On a Linux or macOS system, it's a forward slash (`/`). Java processes both characters identically in a pathname.

When building `Path` objects, the `Path` method `of` can handle the separator characters for you. For example, the statement

Click here to view code image

```
Path documentsPath =
   Path.of(System.getProperty("user.home"), "Documents");
```

creates a `Path` object representing your user account's `Documents` folder. The call

Click here to view code image

```
System.getProperty("user.home")
```

uses the **`System` class's `getProperty` method** to get your user account's folder (represented by `"user.home"`). On a Windows system, this would represent the path

Click here to view code image

```
c:\Users\YourAccount\Documents
```

and on macOS, this would represent the path

```
/Users/YourAccount/Documents
```

We'll use this capability in subsequent examples.

Generative AI

1 In Fig. 11.1, the program terminates immediately after the `DirectoryStream<Path>` elements are displayed, so we did not explicitly close it. Prompt genAIs with the code from Fig. 11.1, asking them to incorporate the `try`-with-resources statement to automatically close the `DirectoryStream<Path>` as soon as the program is done using it.

11.4 Sequential Text Files

Next, we create and manipulate sequential files in which records[3]

are stored in order by the record-key field. We begin with text files, enabling the reader to quickly create and edit human-readable files. We discuss creating, writing data to, reading data from and updating sequential text files. We'll use binary files to read and write images and audio files in **Chapter 19, Building API-Based Java Generative AI Applications**.

3. In this chapter, we use the term "record" to refer to records in a file. Many examples also use `record` classes. When we refer to a `record` class, we format the word "`record`" using a computer font.

11.4.1 Creating a Sequential Text File

Java imposes no structure on a file—notions such as records do not exist as part of the Java language. Therefore, you must structure files to meet the requirements of your programs. The following example shows how to impose a keyed record structure on a file.

The program in this section creates a simple sequential file that might be used in an accounts receivable system to keep track of the amounts owed to a company by its credit clients. For each client, the program obtains from the user an account number and the client's name and balance (i.e., the amount the client owes the company for goods and services received). Each client's data constitutes a record for that client. This program uses the account number as the record key—we will write the file's records in account-number order. In a comprehensive accounts-receivable system (based on sequential files) that includes data input, a sorting capability would be provided so users could enter the records in any order. The records would then be sorted and written to the file.

Class `CreateTextFile`

Class `CreateTextFile` (Fig. 11.2) uses a `Formatter` to output formatted `String`s, using the same formatting features as `System.out.printf`. We broke the program into parts for discussion purposes. Line 8 declares the `record` class `Account` to represent an `Account`'s `accountNumber`, `name` and `balance`—for this chapter's examples, we use only last names.

Click here to view code image

```
1  // Fig. 11.2: CreateTextFile.java
2  // Writing data to a sequential text file with class Formatter.
3  import java.util.Formatter;
4  import java.nio.file.Path;
```

```
5  import java.util.List;
6
7  // record class to represent an account
8  record Account(int accountNumber, String name, double balance) {}
9
```

Fig. 11.2 | Writing data to a sequential text file with class `Formatter`.

`List<Account>` and `List` `static` Method `of`

Section 6.20 introduced the generic class `ArrayList` for creating dynamically resizable array-like data structures. As you'll see in Chapter 12, Generic Collections, the **generic `List` interface** (package `java.util`) specifies the common features for lists of objects, like `ArrayList`s.

Many of this chapter's examples use the same five `Account` objects without changing their data. For such cases, the **`List` `static` method `of`** creates and returns an immutable (unmodifiable) object that implements the `List` interface and contains the method's arguments. Immutable means you cannot add objects to or remove objects from the `List` after you create it. The `List`'s element type is determined from `of`'s arguments. Lines 13–19 create a `List<Account>` containing the five `Account` objects we'll write to the text file in this example.

Click here to view code image

```
10 public class CreateTextFile {
11    public static void main(String[] args) {
12       // account data to write to a text file
13       List<Account> accounts = List.of(
14          new Account(100, "Devi", 24.98),
15          new Account(200, "Taylor", 345.67),
16          new Account(300, "Huber", 0.00),
17          new Account(400, "Ito", -42.16),
18          new Account(500, "Lopez", 224.62)
19       );
20
```

`Path` to the `clients.txt` File

Lines 22–23 create a `Path` object representing the `clients.txt` file. This example writes the file into your user account's `Documents` folder.

Click here to view code image

```
21          // Path to clients.txt in the user's Documents folder
22          Path filePath = Path.of(System.getProperty("user.home"),
23             "Documents", "clients.txt");
24
```

Using a `Formatter` to Write the `Account`s to a File

A `Formatter` object can output to various locations, such as a command window or a file, as we do in this example. Line 26 creates the `Formatter` in the `try`-with-resources statement (introduced in Section 10.12). Recall that `try`-with-resources closes its resource(s) when the `try` block terminates successfully or due to an exception. This ensures we close the file when we no longer need it so other programs can use it. The `Formatter` constructor we use here takes one argument—a `String` containing the file's location and name. We call the `filePath` object's `toString` method to get its `String` representation. If you were to specify only the file's name, the JVM would assume the file is in the directory from which you execute the program. For text files, we use the `.txt` file extension. If the file does not exist, the `Formatter` constructor will create it. If no exception occurs, the file is open for writing, and the `Formatter` object can be used to write data to the file.

Click here to view code image

```
25          // open clients.txt, output data to the file then close clients.txt
26          try (var output = new Formatter(filePath.toString())) {
27             // output each Account to file
28             for (var account : accounts) {
29                output.format("%d %s %.2f%n", account.accountNumber(),
30                   account.name(), account.balance());
31             }
32
33             System.out.printf(
34                "Text file created at: %s%n", filePath.toAbsolutePath());
35          }
36          catch (Exception e) {
37             System.err.printf(
38                "Error writing text file: %s%n", e.getMessage());
39          }
40       }
41    }
```

```
Text file created at: C:\Users\pauldeitel\Documents\clients.txt
```

If no exception occurs while opening the file, lines 28–31 output each `Account`'s data using **`Formatter`'s `format` method**, which works like `System.out.printf` but writes a formatted `String` to the `Formatter` object's output destination (the file `clients.txt`). The format string `"%d %s %.2f%n"` indicates that the current record will be stored as an `int` (the account number) followed by a `String` (the name) and a `double` (the balance). We separate each piece of information from the next by a space and output the `double` value with two digits of precision.

When the `try` block terminates, the `try`-with-resources statement closes the `Formatter` and the underlying output file by calling **`Formatter` method `close`**. If a program does not explicitly call method `close`, the operating system will close the file when program execution terminates—an example of operating-system "termination housekeeping." You should always explicitly close a file when it's no longer needed.

Possible Exceptions

Line 36 catches any `Exception`s that occur in the `try` block. There are several:

- A **`SecurityException`** occurs if the user does not have permission to write data to the file opened in line 26.
- A **`FileNotFoundException`** occurs if the file does not exist and a new file cannot be created or if there's an error opening the file in line 26.
- A **`FormatterClosedException`** occurs if the `Formatter` object is closed when you attempt to use it in lines 29–30 to write into a file.

For these exceptions, we display that there was an error writing the file and show the exception's message.

Caution: Check Whether a File Exists Before Opening It for Writing

In general, you should use the features in Fig. 11.1 to check whether a file exists before you open it for output. Opening an existing file for output **truncates** its contents—all the data in the file is discarded.

Sample Output

This program's output simply indicates that it wrote the `Account`s into `clients.txt`. It does not show how the data records actually appear in the file. In Section 11.4.2, we present a program that reads the file and prints its contents to verify that the file was created successfully. Because this is a text file, you can also verify the information simply by opening the file in a text editor.

Generative AI

1 Prompt genAIs to compare the performance of Java's `BufferedWriter` and `Formatter` for writing large text files and discuss when to use each.
2 Prompt genAIs with Fig. 11.2's code, asking them to rewrite the program to use a `BufferedWriter` rather than a `Formatter`.

11.4.2 Reading Data from a Sequential Text File

We store data in files so we can retrieve it for processing when needed. In Fig. 11.3, we read the data sequentially from the file `"clients.txt"` created by the program of Fig. 11.2. Previously, we've used `Scanner`s to input data from the user. Here, we use one to input each record from the file and display its contents.

Click here to view code image

```
1  // Fig. 11.3: ReadTextFile.java
2  // Reading a text file using a Scanner.
3  import java.nio.file.Path;
4  import java.util.Scanner;
5
6  public class ReadTextFile {
7     public static void main(String[] args) {
8        // Path to clients.txt in user's Documents folder
9        Path filePath = Path.of(System.getProperty("user.home"),
10          "Documents", "clients.txt");
11
12       // open clients.txt, read its contents and close the file
13       try (var input = new Scanner(filePath)) {
14          System.out.printf("%-10s%-12s%10s%n",
15             "Account", "Name", "Balance");
16
17          // read record from file
```

```
18            while (input.hasNext()) { // while there is more to read
19               // display record contents
20               System.out.printf("%-10d%-12s%10.2f%n",
21                  input.nextInt(), input.next(), input.nextDouble());
22            }
23         }
24         catch (Exception e) {
25            System.err.printf(
26               "Error reading file: %s%n", e.getMessage());
27         }
28      }
29   }
```

```
Account   Name      Balance
100       Devi        24.98
200       Taylor     345.67
300       Huber        0.00
400       Ito        -42.16
500       Lopez      224.62
```

Fig. 11.3 | Reading a text file using a `Scanner`.

The `try`-with-resources statement (line 13) opens the file for reading by creating a `Scanner` object. We pass a `Path` object to the constructor, which specifies that the `Scanner` object will read from the file `"clients.txt"` in the user's `Documents` folder.

Lines 14–15 display headers for the columns in the program's output. Lines 18–22 read and display the file's content until end-of-file is reached—in which case, the `hasNext` method will return `false` at line 18. Lines 20–21 use `Scanner` methods `nextInt`, `next` and `nextDouble` to input an `int` (the account number), a `String` (the name) and a `double` (the balance). Each record is one line of text in the file.

The format string in line 20 left-aligns the account number and name and right-aligns the balance with two digits of precision. Each iteration of the loop inputs one line (record) from the text file. When the loop terminates and line 23 is reached, the `try`-with-resources statement implicitly calls the **`Scanner`'s `close` method** to close the `Scanner` and the file.

Potential Exceptions

Several exceptions can occur in the `try` block:

- If the `Scanner` cannot open the file in line 14, an `IOException` occurs.

- If the information in the file is not correctly formatted (e.g., there's a name where there should be a balance), a `InputMismatchException` occurs when the record is input.
- If the `Scanner` is closed before the data is input, an **`IllegalStateException`** occurs.

These exceptions are handled by lines 25–28.

Generative AI

1 Prompt genAIs to compare the performance of Java's `BufferedReader` and `Scanner` for reading large text files and discuss when to use each.
2 Prompt genAIs with Fig. 11.3's code asking them to rewrite it to use `BufferedReader` rather than `Scanner`.

11.4.3 Updating Sequential Files

The data in many sequential files cannot be modified without the risk of destroying other data in the file. For example, if you need to change the name "`Pappas`" to "`Papadopoulou`", the old name cannot simply be overwritten, because the new name requires more space. The record for `Pappas` was written to the file as

```
300 Pappas 0.00
```

If you were to rewrite the record beginning at the same location in the file using the new name, the record would be

```
300 Papadopoulou 0.00
```

The new record is larger (has more characters) than the original record. "`Papadopoulou`" would overwrite the "`0.00`" in the current record, and the characters beyond the last "`u`" in "`Papadopoulou`" would overwrite the beginning of the next sequential record in the file.

The problem here is that fields in a text file—and hence records—can vary in size. For example, 7, 14, –117, 2074 and 27383 are all `int`s stored in the same number of bytes (4) internally, but they're different-sized fields when written to a file as text. Therefore, you do not usually update records in a sequential file in place—instead, you rewrite the entire file. To make the preceding

name change:

- the records before `300 Pappas 0.00` would be copied to a new file,
- the new record (which can be of a different size than the one it replaces) would be written and
- the records after `300 Pappas 0.00` would be copied to the new file.

Rewriting the entire file is uneconomical to update just one record, but reasonable if a substantial number of records need to be updated.

Generative AI

1 Prompt genAIs, asking what Java techniques you can use to optimize the process of rewriting sequential files for record updates.

11.5 Case Study: JavaScript Object Notation (JSON) Serialization

More and more computing today is done "in the cloud"—that is, distributed across the Internet. Many applications you use daily communicate over the Internet with **cloud-based services** that use massive clusters of computing resources (computers, processors, memory, storage, databases, etc.).

A service that provides access to itself over the Internet is known as a **web service**. Applications typically communicate with web services by sending and receiving **JavaScript Object Notation (JSON)** objects. JSON is a text-based, human- and computer-readable data-interchange format that represents objects as collections of name-value pairs. JSON has become the preferred data format for transmitting objects between platforms over the Internet. In this case study, we'll explore using JSON in Java.

JSON Data Format

Each JSON object contains a comma-separated list of property

names and values in curly braces. For example, the following name–value pairs might represent a client record:

Click here to view code image

```
{"accountNumber": 100, "name": "Devi", "balance": 24.98}
```

JSON also supports arrays as comma-separated values in square brackets. For example, the following represents a JSON array of numbers:

```
[100, 200, 300]
```

Values in JSON objects and arrays can be

- strings in double quotes (like `"Devi"`),
- numbers (like `100` or `24.98`),
- JSON Boolean values (represented as `true` or `false`),
- `null` (to represent no value),
- arrays of any valid JSON value and
- other JSON objects containing values and arrays of the types mentioned above.

JSON arrays may contain elements of the same or different types.

Serialization

Converting an object into a format suitable for storage or transmission over the Internet is known as **serialization**. Similarly, reconstructing an object from serialized data is known as **deserialization**. JSON is just one of several serialization formats. Another common one is XML (eXtensible Markup Language). JSON is generally preferred for its simplicity (it's less verbose than XML), ease of use, efficiency and widespread adoption.[4,5,6,7]

4. Hackr.io. "JSON vs XML in 2024: Comparing Features and Examples." Accessed November 29, 2024. https://hackr.io/blog/json-vs-xml.
5. W3Schools. "JSON vs XML." Accessed November 29, 2024. https://www.w3schools.com/js/js_json_xml.asp.
6. Imaginary Cloud. "JSON vs XML: Which One Is Faster and More Efficient?" Accessed November 29, 2024. https://www.imaginarycloud.com/blog/json-vs-xml.
7. Coursesity. "JSON vs XML Comparison: Which Is Better? [2024]." Accessed November 29, 2024. https://coursesity.com/blog/json-

vs-xml/.

Serialization Security

Some programming languages (including Java) have serialization mechanisms that use a language-native format. Deserializing objects using these native formats is a source security issues. According to the Open Web Application Security Project (OWASP), these "can be repurposed for malicious effect when operating on untrusted data. Attacks against deserializers have been found to allow denial-of-service, access control, and remote code execution (RCE) attacks." OWASP indicates that you can significantly reduce attack risk by avoiding language-native serialization formats in favor of "pure data" formats like JSON or XML.[8]

8. "Deserialization Cheat Sheet." OWASP Cheat Sheet Series. Accessed November 11, 2024. https://cheatsheetseries.owasp.org/cheatsheets/Deserialization_Cheat_Sheet.html.

Jackson Open-Source Library

The **Jackson open-source library**[9] is a popular Java library for converting objects to JSON and vice versa with minimal code. Jackson was initially developed as a JSON processing library but has since evolved into a multi-format data-processing library. Later in this chapter, we'll also use it to process comma-separated value (CSV) files. You can access Jackson's API documentation and download the library from the following links:

9. "Jackson Project Home @github." Accessed November 11, 2024. https://github.com/FasterXML/jackson.

Click here to view code image

```
https://javadoc.io/doc/com.fasterxml.jackson.core/jackson-core/latest

https://github.com/FasterXML/jackson
```

For basic JSON and CSV processing with the Jackson library, you typically need the following Jackson library components:

- **jackson-core**: Provides the core capabilities for parsing and generating JSON.
- **jackson-databind**: Provides capabilities for converting Java objects to and from JSON.
- **jackson-annotations**: Provides Jackson-specific annotations

used to customize serialization and deserialization.

- **`jackson-dataformat-csv`**: Provides additional capabilities for reading and writing CSV files.

For your convenience, we've included these components in the subfolder `libs` with this chapter's examples. After discussing Fig. 11.4, we show the `javac` and `java` commands required to compile and run the Jackson-based examples.

Generative AI

1 Prompt genAIs asking them to discuss in the context of Java why serialization and deserialization are more dangerous with binary files than with text files, such as files containing JSON-formatted text.

11.5.1 Serializing a `List<Account>`

Figure 11.4 reimplements Fig. 11.2 using JSON serialization. Jackson's core functionality is provided by its **`ObjectMapper` class** (imported at line 3), which provides methods to serialize Java objects to JSON and deserialize JSON to Java objects.

Click here to view code image

```
1  // Fig. 11.4: CreateJSONFile.java
2  // Writing data to a JSON file using the Jackson open-source library.
3  import com.fasterxml.jackson.databind.ObjectMapper;
4  import java.io.IOException;
5  import java.nio.file.Path;
6  import java.util.List;
7
8  // record class to represent an account
9  record Account(int accountNumber, String name, double balance) {}
10
11 public class CreateJSONFile {
12    public static void main(String[] args) {
13       // account data to write to JSON
14       List<Account> accounts = List.of(
15          new Account(100, "Devi", 24.98),
16          new Account(200, "Taylor", 345.67),
17          new Account(300, "Huber", 0.00),
18          new Account(400, "Ito", -42.16),
19          new Account(500, "Lopez", 224.62)
20       );
```

```
21
22          // Path to clients.json in user's Documents folder
23          Path filePath = Path.of(System.getProperty("user.home"),
24             "Documents", "clients.json");
25
26          var mapper = new ObjectMapper(); // performs the serialization
27
28          try {
29             // Write accounts to a JSON file
30             mapper.writeValue(filePath.toFile(), accounts);
31             System.out.printf(
32                "JSON file created at: %s%n", filePath.toAbsolutePath());
33          }
34          catch (IOException e) {
35             System.err.printf(
36                "Error writing JSON file: %s%n", e.getMessage());
37          }
38       }
39    }
```

```
JSON file created at: C:\Users\pauldeitel\Documents\clients.json
```

Fig. 11.4 | Writing data to a JSON file using the Jackson open-source library.

Line 26 creates an `ObjectMapper`. Line 30 uses **`ObjectMapper`'s `writeValue` method** to serialize a Java object and write it directly to a file:

- The first argument `filePath.toFile()` converts the `Path` object representing the file location into a `File` object in which `writeValue` writes the serialized data.
- The second is the Java object to serialize—in this case, the `List<Account>` named `accounts`.

Jackson automatically converts the `List<Account>` into a JSON array in square brackets (`[` and `]`) in which each `Account` object is serialized as a JSON object in curly braces (`{` and `}`).

`clients.json` Contents

The resulting `clients.json` file contains the following JSON representation of the `List`. We reformatted the JSON for readability. Even though you may never have seen JSON-encoded data, you should find this to be well organized, readable and pretty understand-able—these are important reasons for JSON's appeal.

Click here to view code image

```
[
   {
      "accountNumber": 100,
      "name": "Devi",
      "balance": 24.98
   },
   {
      "accountNumber": 200,
      "name": "Taylor",
      "balance": 345.67
   },
   {
      "accountNumber": 300,
      "name": "Huber",
      "balance": 0.0
   },
   {
      "accountNumber": 400,
      "name": "Ito",
      "balance": -42.16
   },
   {
      "accountNumber": 500,
      "name": "Lopez",
      "balance": 224.62
   }
]
```

Requirements for Serializable Classes

By default, Jackson serializes all of a `record`'s instance variables. For classes, Jackson requires *get* methods for each instance variable to serialize. When deserializing JSON to a `record` object, Jackson uses the `record`'s constructor to reconstruct the object. For other classes, it requires a default constructor and *set* methods to store each instance variable in the new object.

Compiling and Running the Program

For your convenience, we included the Jackson library components that this chapter's examples require in the subfolder `libs` with Chapter 11's examples. To compile this example, use the following command on Windows:

Click here to view code image

```
javac -cp ".;../libs/*" CreateJSONFile.java
```

For macOS, change the semicolon (`;`) to a colon (`:`). The **`-cp` option** specifies the **class-path**, which indicates where the classes this program needs to compile can be found. The dot (`.`) indicates the current folder. Depending on your operating system, the semicolon (`;`) or colon (`:`) separates one location from another. The relative path `../libs/*` specifies that the `libs` folder is up one level in the folder structure (i.e., in the `ch11` folder containing this chapter's examples). Finally, the `*` indicates that all the files in that folder should be added to the classpath. To run this example, use the following command on Windows:

Click here to view code image

```
java -cp ".;../libs/*" CreateJSONFile
```

For macOS, change the semicolon (`;`) to a colon (`:`).

Generative AI

1 Prompt genAIs with the code in Fig. 11.4, asking them to update it so Jackson writes the JSON data with indentation to make the output file more readable. Run the updated program, then open `clients.json` in a text editor to see the result.

11.5.2 Deserializing a `List<Account>`

Figure 11.5 reimplements Fig. 11.3 using JSON deserialization to create a `List<Account>` representing the account data in the file. For Jackson to recreate objects of regular Java classes in memory, it must have access to each class's default or no-argument constructor. It will use that to create an object then use that object's *set* methods to place the data into the object. For a `record` class, Jackson uses the autogenerated constructor.

Click here to view code image

```
1  // Fig. 11.5: ReadJSONFile.java
2  // Reading data from a JSON file using the Jackson open-source library.
3  import com.fasterxml.jackson.core.type.TypeReference;
4  import com.fasterxml.jackson.databind.ObjectMapper;
5  import java.io.IOException;
6  import java.nio.file.Path;
7  import java.util.List;
```

```
8
9  // record class to represent an account
10 record Account(int accountNumber, String name, double balance) {}
11
12 public class ReadJSONFile {
13    public static void main(String[] args) {
14       // Path to clients.json in user's Documents folder
15       Path filePath = Path.of(System.getProperty("user.home"),
16          "Documents", "clients.json");
17
18       var mapper = new ObjectMapper(); // performs the deserialization
19
20       try {
21          // Read accounts from the JSON file
22          List<Account> accounts = mapper.readValue(
23             filePath.toFile(), new TypeReference<List<Account>>() {});
24
25          // display the accounts
26          System.out.printf(
27             "%-10s %-10s %7s%n", "Account", "Name", "Balance");
28
29          for (var account : accounts) {
30             System.out.printf("%-10d %-10s %7.2f%n",
31                account.accountNumber(), account.name(),
32                account.balance());
33          }
34       }
35       catch (IOException e) {
36          System.err.printf(
37             "Error reading JSON file: %s%n", e.getMessage());
38       }
39    }
40 }
```

```
Account    Name       Balance
100        Devi         24.98
200        Taylor      345.67
300        Huber         0.00
400        Ito         -42.16
500        Lopez       224.62
```

Fig. 11.5 | Reading data from a JSON file using the Jackson open-source library.

Lines 22–23 use the **`ObjectMapper`'s `readValue` method** to deserialize the JSON data from the file in its first argument into a `List<Account>`. Jackson needs to know the type of data you want to convert (in this case, a `List<Account>`), which is the purpose of method `readValue`'s second argument:

Click here to view code image

```
new TypeReference<List<Account>>() {}
```

This uses Jackson's **`TypeReference` class** (imported at line 3) to provide Jackson with the type information the `ObjectMapper` needs to deserialize complex generic types (such as a `List<Account>`), mapping the JSON data to the correct Java object structure. This expression creates an object of an **anonymous inner class**—a class with no name that you create "on-the-fly" to extend another class or implement an interface. Such a class exists only in the place where it's written. In the preceding expression, `TypeReference` is an `abstract` class from the Jackson library that helps it understand generic types like `List<Account>`. The expression creates a new unnamed class that extends

```
TypeReference<List<Account>>
```

and immediately creates one object of this new unnamed class. Jackson uses this object to help it deserialize your data into a `List<Account>`. The anonymous inner class is similar to the following class declaration:

Click here to view code image

```
class MyTypeReference extends TypeReference<List<Account>> {
   // empty class body
}
```

Once the data is deserialized, we display it using the same format as in Fig. 11.3.

Generative AI

1 Prompt genAIs to write a brief tutorial on Java anonymous inner class fundamentals.
2 Prompt genAIs for a brief tutorial on why you might want to customize Jackson JSON serialization and deserialization and how to do it.

11.6 Case Study: Processing a JSON Response from a Web Service

Section 1.8 introduced the Internet, the web, the cloud, web services and mashups. In this case-study, you'll dive into the world of web services using Java's built-in networking capabilities to invoke a web service and process the JSON results it returns. You'll interact with an OpenWeatherMap free web service from

```
https://openweathermap.org/
```

to get the current weather report for a city of your choosing, which you'll specify with the city name as a command-line argument to the application.

Web Services

The machine on which a web service resides is referred to as a **web-service host**. A **web-service client application** (in our case, a Java program) sends a **web-service request** over a network to the web-service host, which processes the request and returns a **web-service response** over the network to the client. This kind of distributed computing benefits systems in various ways. For example, an application without direct access to data on another system might be able to retrieve the data via a web service. Similarly, an application lacking the processing power to perform specific computations could use a web service to take advantage of another system's superior resources.

Representational State Transfer (REST)

Most web services today use **Representational State Transfer (REST)** and are often called **RESTful web services**. In a RESTful web service, each function you can call is identified by a unique URL. **URLs (Uniform Resource Locators)** identify the locations of resources on the Internet, such as websites and web services. For example, `https://deitel.com/` is the URL for the Deitel & Associates website. When a web server receives a request to a RESTful web service, it immediately knows what function to call on that server. RESTful web services can be called from programs, as you'll do here, or directly from a web browser's address bar by entering the appropriate URL.

OpenWeatherMap

OpenWeatherMap provides a free tier to several of its weather web

services. Before using them, you must sign up for a free account at https://openweathermap.org/. They will send you an email to verify your account. Once you do, they'll reply with an email that contains your free API key. You can also find this under the API Keys tab in your account when you log into the site.

You can view the variety of OpenWeatherMap APIs and their documentation at

Click here to view code image

```
https://openweathermap.org/api
```

Some are free, and some are available only to subscribers. With a free API key, you can access:

- the Current Weather Data for a specified location, which we use in this case study,
- the One Call API, which returns a combination of current and future weather data for a specified location, and
- the 5-Day/3-Hour Forecast for a specified location.

Storing the API Key in an Environment Variable

The program in this section assumes you store your API key in a system environment variable named `OPENWEATHERMAP_API_KEY`. On Windows, perform the following steps to store your API key in an environment variable:

- Enter `"environment"` in the taskbar's Search field.
- In the search results, select Edit the system environment variables.
- In the System Properties dialog, select or press the Environment Variables button.
- Under User variables, click New….
- Enter the Variable name `OPENWEATHERMAP_API_KEY` then, for the Variable value, paste in the API key provided by OpenWeatherMap.org.
- Click OK to save the environment variable, then click OK in the System Properties dialog to close it.

On macOS, perform the following steps to store your API key in an environment variable:

- Open Terminal.
- Open `~/.zshrc` using a text editor (e.g., `nano ~/.zshrc`).
- Scroll to the end of the file and add the following line, replacing *YourAPIKey* with the API key provided by OpenWeatherMap.org:

  Click here to view code image

  ```
  export OPENWEATHERMAP_API_KEY="YourAPIKey"
  ```

- Save and close the file.
- Run the following command in the Terminal:

  ```
  source ~/.zshrc
  ```

JSON and Web Services

Many cloud-based services like OpenWeatherMap communicate with your applications via JSON objects. Figure 11.6 contains a sample JSON response from OpenWeatherMap's Current Weather Data web service, which we formatted for readability. The items we highlighted in **`bold`** contain the information we'll display in this example. We'll direct Jackson to ignore the non-bold items.

Click here to view code image

```
{
  "coord": {
    "lon": -71.0598,
    "lat": 42.3584
  },
  "weather": [
    {
      "id": 802,
      "main": "Clouds",
      "description": "scattered clouds",
      "icon": "03d"
    }
  ],
  "base": "stations",
  "main": {
    "temp": 6.55,
    "feels_like": 2.61,
    "temp_min": 5.53,
    "temp_max": 8.06,
    "pressure": 1009,
    "humidity": 57,
```

```
      "sea_level": 1009,
      "grnd_level": 1005
   },
   "visibility": 10000,
   "wind": {
      "speed": 6.69,
      "deg": 240,
      "gust": 12.35
   },
   "clouds": {
      "all": 40
   },
   "dt": 1732901160,
   "sys": {
      "type": 2,
      "id": 2013408,
      "country": "US",
      "sunrise": 1732881140,
      "sunset": 1732914812
   },
   "timezone": -18000,
   "id": 4930956,
   "name": "Boston",
   "cod": 200
}
```

Fig. 11.6 | Sample OpenWeatherMap response for Boston, MA, USA.

Getting a Weather Report

Figure 11.7 interacts with OpenWeatherMap to get a weather report for the city the user specifies as a command-line argument to the program. You can compile this program as you did with the programs in Section 11.5. To get a weather report for Boston like the one in Fig. 11.6, you would execute the program on Windows with

Click here to view code image

```
java -cp ".;../libs/*" WeatherApp Boston
```

On macOS, replace the semicolon (`;`) with a colon (`:`). We broke the program into parts for discussion purposes.

Click here to view code image

```
1  // Fig. 11.7: WeatherApp.java
```

```
2  // Use an OpenWeatherMap web service to get a city's current weather.
3  import com.fasterxml.jackson.annotation.JsonIgnoreProperties;
4  import com.fasterxml.jackson.databind.ObjectMapper;
5
6  import java.net.URI;
7  import java.net.URLEncoder;
8  import java.net.http.HttpClient;
9  import java.net.http.HttpRequest;
10 import java.net.http.HttpResponse;
11 import java.nio.charset.StandardCharsets;
12 import java.time.Instant;
13 import java.time.ZoneId;
14 import java.time.ZoneOffset;
15 import java.time.format.DateTimeFormatter;
16
```

Fig. 11.7 | Use an OpenWeatherMap web service to get a city's current weather.

`record` Classes for Deserializing the JSON Response

The record classes `WeatherData` (lines 18–25), `Weather` (lines 27–32) and `Main` (lines 34–40) mimic the structure of the JSON data we'll extract from the web service response. The entire response in Fig. 11.6 represents the weather data for a city. Class `WeatherData` contains instance variables named for each top-level JSON key (Fig. 11.6, lines 6, 15, 34, 42 and 44) we use in this example. We've used the compile-time annotation `@Override` in previous chapters. Many libraries use runtime annotations to inject code into your programs and perform tasks for you. Line 18 above class `WeatherData` uses the Jackson runtime annotation `@JsonIgnoreProperties` to tell Jackson to ignore the rest of the top-level JSON fields from Fig. 11.6 that we do not use in this example.

Click here to view code image

```
17 // record classes for JSON mapping
18 @JsonIgnoreProperties(ignoreUnknown = true)
19 record WeatherData(
20    Weather[] weather,
21    long dt, // date and time
22    Main main,
23    int timezone,
24    String name
25 ) {}
26
```

The JSON keys `dt`, `timezone` and `name` are each single values that map to the corresponding `WeatherData` instance variables. The JSON key `weather` (Fig. 11.6, line 6) has as its value a JSON array of JSON objects—in Fig. 11.6, the array contains one JSON object. Similarly, the JSON key `main` (Fig. 11.6, line 15) has as its value a JSON object containing many name-value pairs. So, our `WeatherData` class declares instance variable `weather` as an array of `Weather` objects (defined in lines 27–32 below) and instance variable `main` as an object of class `Main` (defined in lines 34–40 below). We named these two classes based on the JSON keys they represent. Again, we do not use all the JSON name-value pairs, so both classes use the Jackson annotation `@JsonIgnoreProperties` to ignore the name-value pairs that do not match these classes' instance variables.

Click here to view code image

```
27  @JsonIgnoreProperties(ignoreUnknown = true)
28  record Weather(
29     String main,
30     String description,
31     String icon
32  ) {}
33
34  @JsonIgnoreProperties(ignoreUnknown = true)
35  record Main(
36     double temp,
37     double feels_like,
38     int pressure,
39     int humidity
40  ) {}
41
```

Getting the City from the Command-Line Argument

Lines 45–50 check if there is a command-line argument for the city and, if so, assign it to the `String city`.

Click here to view code image

```
42  public class WeatherApp {
43     public static void main(String[] args) throws Exception {
44        // Check for proper command-line arguments
45        if (args.length != 1) {
46           System.out.println("Usage: WeatherApp <city>");
47           System.exit(1);
48        }
```

```
49
50          String city = args[0]; // city from command-line argument
51
```

Getting the API Key

Line 53 uses **`System`'s `static` method `getenv`** to load the API key from the environment variable `OPENWEATHERMAP_API_KEY`. Lines 56–60 ensure the API key is not null or empty.

Click here to view code image

```
52          // load API key from OPENWEATHERMAP_API_KEY environment variable
53          String apiKey = System.getenv("OPENWEATHERMAP_API_KEY");
54
55          // check if API key exists and is set
56          if (apiKey == null || apiKey.isBlank()) {
57             System.out.println(
58                "OPENWEATHERMAP_API_KEY environment variable is not set.");
59             System.exit(1);
60          }
61
```

Creating the URL that Invokes the Web Service

Lines 63–70 construct a valid URL for requesting weather data from the OpenWeatherMap web service. **`URLEncoder` method `encode`** converts the `city` variable into a format safe for use in a URL—in case the command-line argument contains characters not allowed in URLs. Next, we insert the `encodedCity` into the URL, which specifies

- the web service API endpoint (`https://api.openweathermap.org/data/2.5/weather`),
- the city name, which replaces `%s` in `q=%s`,
- the temperature units (`units=metric`), and
- the required API key, which replaces `%s` in `appid=%s`

The API endpoint is followed by `?` and the options, each separated from the next by `&`.

Click here to view code image

```
62          // URL encode the city parameter
63          String encodedCity =
```

```
64             URLEncoder.encode(city, StandardCharsets.UTF_8);
65
66          // create the API URL as a String
67          String apiUrl = String.format(
68              "https://api.openweathermap.org/" +
69              "data/2.5/weather?q=%s&units=metric&appid=%s",
70              encodedCity, apiKey);
71
```

Setting up the Web Service Request and Receiving the Response

To obtain the JSON response in Fig. 11.6, our application uses the **`HttpClient` class** (package **`java.net.http`**)[10] to make a request to the OpenWeatherMap web service.

10. "JEP 321: HTTP Client (Standard)." Accessed January 1, 2025. https://openjdk.org/jeps/321.

Click here to view code image

```
72          // use Java's HttpClient to make the request
73          HttpClient client = HttpClient.newHttpClient();
74
75          // build the request that HttpClient will use to
76          // invoke the OpenWeatherMap web service
77          HttpRequest request =
78             HttpRequest.newBuilder()
79                .uri(URI.create(apiUrl))
80                .GET()
81                .build();
82
83          // send the HTTP request and get the response
84          HttpResponse<String> response =
85             client.send(request, HttpResponse.BodyHandlers.ofString());
86
87          // check if the request was successful
88          if (response.statusCode() != 200) {
89             System.out.printf("Error: Received HTTP status %d\n",
90                response.statusCode());
91             System.exit(1);
92          }
93
```

Class `HttpClient` encapsulates the mechanisms to request information from web services and receive their responses. Line 73 calls the class's **`HttpClient`'s `static` method `newHttpClient`** to get an `HttpClient` object. Next, lines 77–81 use a chained method call

to build an `HttpRequest`:

- Line 78 calls **`static` method `newBuilder`** to get an **`HttpRequest.Builder`** object for configuring the request.
- Line 79 calls the builder's **`uri` method** to specify where the web service resides on the Internet using the `apiUrl` created in lines 67–70. A **Uniform Resource Identifier (URI)** is a more general form of the Uniform Resource Locators (URLs) used to locate websites. **`URI` method `create`** converts the URL to a URI.
- Line 80 uses the **`GET` method** to set the HTTP request method to GET, which indicates that the program will request data.
- Line 81 calls the builder's **`build` method** to create the `HttpRequest` object.

Next, lines 84–85 send the request using the `HttpClient` object's `send` method:

- The first argument is the `HttpRequest`.
- The second argument uses the **`HttpResponse.BodyHandlers` class's `ofString` method** to indicate that we expect the response to be a `String`.

The response is stored in an `HttpResponse<String>` object, which holds the status code that indicates whether the request was successful and, if so, the response content. Line 88 checks the status code—the status code 200 confirms the server processed the request successfully. Otherwise, the program prints an error message with the received status code and terminates. For more information on HTTP status codes, visit `https://en.wikipedia.org/wiki/List_of_HTTP_status_codes`.

Deserializing the Response into a `WeatherData` Object

Lines 95–97 use the techniques you used in Section 11.5 to deserialize the JSON weather report in the `HttpResponse`'s body and store it in a `WeatherData` object that mimics the structure of the JSON response. Here, we use `WeatherData.class` to represent the object's type, rather than `TypeReference` from Fig. 11.5—`TypeReference` is used specifically to provide Jackson with additional information about generic types.

Click here to view code image

```
94          // parse the JSON response directly into the WeatherData record
95          ObjectMapper mapper = new ObjectMapper();
96          WeatherData weatherData =
97             mapper.readValue(response.body(), WeatherData.class);
98
```

Preparing and Displaying the Weather Report

Lines 100–102 convert the Celsius temperatures to Fahrenheit so we can display both:

Click here to view code image

```
99          // convert temperatures to Fahrenheit
100         double tempCurrentF = (weatherData.main().temp() * 9 / 5) + 32;
101         double feelsLikeF =
102            (weatherData.main().feels_like() * 9 / 5) + 32;
103
```

Line 105 gets the weather report's icon from the `weatherData` object's `weather` instance variable, which is an array containing one `Weather` object in this example. Lines 106–107 create a URL for the weather icon—once you study JavaFX in Chapters 15–17, you could use this URL to display the weather icon.

Click here to view code image

```
104         // generate the icon URL
105         String iconName = weatherData.weather()[0].icon();
106         String iconUrl = String.format(
107            "https://openweathermap.org/img/wn/%s@4x.png", iconName);
108
```

Lines 110–115 use capabilities of the Java Date/Time API to convert the weather report's date and time to a readable format. Lines 113–114 indicate that the `DateTimeFormatter` should convert the time to local time, based on the user's time zone:

Click here to view code image

```
109         // format the date and time based on the response
110         Instant timestamp = Instant.ofEpochSecond(weatherData.dt());
111         String formattedDateTime =
112            DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss")
113               .withZone(ZoneId.ofOffset("UTC",
```

```
114                 ZoneOffset.ofTotalSeconds(weatherData.timezone())))
115             .format(timestamp);
116
```

Finally, lines 117–129 display the weather report, sometimes using chained method calls to navigate to the specific item to display. For example, in line 120

```
weatherData.main().temp()
```

gets the `weatherData` object's `main` object by calling `main()`, then gets that object's `temp` instance variable value by calling `temp()`.

Click here to view code image

```
117         System.out.printf("%s Weather\n", weatherData.name());
118         System.out.printf("Date and Time: %s\n", formattedDateTime);
119         System.out.printf("Temperature: %.1f C (%.1f F)\n",
120            weatherData.main().temp(), tempCurrentF);
121         System.out.printf("Feels like: %.1f C (%.1f F)\n",
122            weatherData.main().feels_like(), feelsLikeF);
123         System.out.printf("Pressure: %d hPa\n",
124            weatherData.main().pressure());
125         System.out.printf("Humidity: %d%%\n",
126            weatherData.main().humidity());
127         System.out.printf("Conditions: %s\n",
128            weatherData.weather()[0].description());
129         System.out.printf("Icon: %s\n", iconUrl);
130      }
131   }
```

```
Boston Weather
Date and Time: 2024-12-28 10:18:25
Temperature: 3.8 C (38.9 F)
Feels like: 1.5 C (34.6 F)
Pressure: 1026 hPa
Humidity: 89%
Conditions: light rain
Icon: https://openweathermap.org/img/wn/10d@4x.png
```

Generative AI

1 Prompt genAIs with `WeatherApp.java`, asking them to show how to use Jackson annotations to make the `WeatherData` class's instance variable name `dt` more readable.

2 Prompt genAIs for a brief tutorial on RESTful web services.

3 Prompt genAIs for a brief tutorial on how HTTP GET requests interact with web services to obtain results.
4 Prompt genAIs, asking for a list of special characters used in URLs and why you should use `URLEncode` method `encode` to include data in a URL.
5 Prompt genAIs, asking them to compare and contrast URIs and URLs.
6 Prompt genAIs to create a curated list of free web services and suggest applications you can build by combining them.

11.7 Case Study: Creating and Reading CSV Files

The **CSV (comma-separated values)** file format, which uses the **`.csv` file extension**, is popular for datasets used in big data, data analytics and data science, and in artificial intelligence applications like machine learning and deep learning. Here, we'll demonstrate writing to and reading from a CSV file.

Datasets

There's an enormous variety of free datasets available online. The OpenML machine learning resource site

```
https://openml.org
```

contains almost 6000 free datasets in CSV format. Other great sources of datasets include:

Click here to view code image

```
https://github.com/awesomedata/awesome-public-datasets
https://vincentarelbundock.github.io/Rdatasets/articles/data.html
```

Also, many local and national governments publish free datasets. For example, as of December 2024, the U.S. government listed 25,515 CSV format datasets on `data.gov`:

Click here to view code image

```
https://catalog.data.gov/dataset/?res_format=CSV
```

account.csv

For our two examples, we'll create the file `clients.csv` with the following data

```
accountNumber,name,balance
100,Devi,24.98
200,Taylor,345.67
300,Huber,0.0
400,Ito,-42.16
500,Lopez,224.62
```

then we'll read that file. The first row of a CSV file typically contains column names. Each subsequent row contains one data record representing the values for those columns. This dataset has three columns representing an `accountNumber`, `name` and `balance`.

11.7.1 Creating a CSV File

Figure 11.8 reimplements Fig. 11.4 using CSV format.

Click here to view code image

```
1  // Fig. 11.8: CreateCSVFile.java
2  // Writing data to a CSV file using the Jackson open-source library.
3  import com.fasterxml.jackson.databind.ObjectWriter;
4  import com.fasterxml.jackson.dataformat.csv.CsvMapper;
5  import com.fasterxml.jackson.dataformat.csv.CsvSchema;
6  import java.io.IOException;
7  import java.nio.file.Path;
8  import java.util.List;
9
10 // record class to represent an account
11 record Account(int accountNumber, String name, double balance) {}
12
13 public class CreateCSVFile {
14    public static void main(String[] args) {
15       // account data to write to
16       List<Account> accounts = List.of(
17          new Account(100, "Devi", 24.98),
18          new Account(200, "Taylor", 345.67),
19          new Account(300, "Huber", 0.00),
20          new Account(400, "Ito", -42.16),
21          new Account(500, "Lopez", 224.62)
22       );
23
24       // Path to clients.csv in user's Documents folder
25       Path filePath = Path.of(System.getProperty("user.home"),
```

```
26            "Documents", "clients.csv");
27
28         // create Jackson CsvMapper and Schema
29         var mapper = new CsvMapper();
30         CsvSchema schema =
31            mapper.schemaFor(Account.class) // determines CSV files's fields
32               .withHeader(); // first line in clients.csv is the field names
33
34         try {
35            // ObjectWriter writes data to CSV file using specified schema
36            ObjectWriter writer = mapper.writer(schema);
37            writer.writeValue(filePath.toFile(), accounts);
38            System.out.printf(
39               "CSV file created at: %s%n", filePath.toAbsolutePath());
40         }
41         catch (IOException e) {
42            System.err.printf(
43               "Error writing CSV file: %s%n", e.getMessage());
44         }
45      }
46   }
```

```
CSV file created at: C:\Users\pauldeitel\Documents\clients.csv
```

Fig. 11.8 | Writing data to a CSV file using the Jackson open-source library.

Remember to add Jackson to the classpath when compiling and running this program. The primary differences in this example are Jackson's CSV handling in lines 29–32 and 36–37:

- Line 29 creates a **`CsvMapper`** object. This is similar to the `ObjectMapper` used for JSON but converts Java objects into CSV format and vice versa.
- Lines 30–32 use chained method calls to build a **`CsvSchema`** for class `Account`. This defines the CSV file's structure based on `Account`'s instance variables. The **`CsvSchema` method `schemaFor`** determines the schema from `Account.class`—a shorthand for a Java object that represents the `Account` class's definition. **Method `withHeader`** specifies that the CSV file's first line should contain `Account`'s instance variable names as column headers for the CSV records' fields. By default, a CSV record's fields are separated by commas, but Jackson allows you to customize this to use other separators.
- An **`ObjectWriter`** writes the header and data to the CSV file. Line 36 creates an `ObjectWriter` configured with the

`CsvSchema` from lines 30–32.

- Finally, line 37 uses the **`ObjectWriter`'s `writeValue` method** to write the `List` of `Account`s to the file at the specified path. This method serializes the accounts list into CSV format, producing a file with rows corresponding to each `Account` object and columns matching the fields in the schema.

Generative AI

1 Prompt genAIs for a discussion of using commas vs. custom separators with Jackson.

11.7.2 Reading a CSV File

Figure 11.9 reimplements Fig. 11.5 to read the CSV data, then display it in the same format as Fig. 11.5. Remember to add Jackson to the classpath when compiling and running this program.

Click here to view code image

```
1  // Fig. 11.9: ReadCSVFile.java
2  // Reading data from a CSV file using the Jackson open-source library.
3  import com.fasterxml.jackson.databind.MappingIterator;
4  import com.fasterxml.jackson.dataformat.csv.CsvMapper;
5  import com.fasterxml.jackson.dataformat.csv.CsvSchema;
6
7  import java.io.IOException;
8  import java.nio.file.Path;
9  import java.util.List;
10
11 // record class to represent an account
12 record Account(int accountNumber, String name, double balance) {}
13
14 public class ReadCSVFile {
15    public static void main(String[] args) {
16       // Path to clients.csv in user's Documents folder
17       Path filePath = Path.of(System.getProperty("user.home"),
18          "Documents", "clients.csv");
19
20       var mapper = new CsvMapper(); // reads CSV records
21
22       // specify that Jackson should map each CSV column to
23       // an instance variable in the record class Account
24       CsvSchema schema =
25          mapper.schemaFor(Account.class) // fields map to Account objects
26                .withHeader(); // first line of text is the field names
```

```
27
28          // read the CSV file
29          try {
30             // MappingIterator provides access to records as Account objects
31             MappingIterator<Account> iterator =
32                mapper.readerFor(Account.class).with(schema).readValues(
33                   filePath.toFile());
34
35             List<Account> accounts = iterator.readAll();
36
37             // display the accounts
38             System.out.printf(
39                "%-10s %-10s %7s%n", "Account", "Name", "Balance");
40
41             for (var account : accounts) {
42                System.out.printf("%-10d %-10s %7.2f%n",
43                   account.accountNumber(), account.name(),
44                   account.balance());
45             }
46          }
47          catch (IOException e) {
48             System.err.printf(
49                "Error reading CSV file: %s%n", e.getMessage());
50          }
51       }
52    }
```

```
Account    Name       Balance
100        Devi         24.98
200        Taylor      345.67
300        Huber         0.00
400        Ito         -42.16
500        Lopez       224.62
```

Fig. 11.9 | Reading data from a CSV file using the Jackson open-source library.

The primary differences in this example are Jackson's CSV handling in lines 20, 24–26 and 31–35:

- Line 20 creates a `CsvMapper` object that will convert CSV records into Java objects.
- Lines 24–26 create a `CsvSchema` for the `Account` class. This schema defines how the columns in the CSV file correspond to the instance variables in an `Account`. As in Fig. 11.8, we indicate that the CSV file's first row contains column headers. Jackson uses these to map the columns to the corresponding

fields in the `Account record` object.

- To read the CSV file, we use a **`MappingIterator`**. Lines 31–33 create a `Mapping-Iterator<Account>` using a chained method call. An "iterator" is an object that enables a program to iterate over a collection's elements one by one. For example, an enhanced `for` loop uses an iterator to access each element in a collection, such as an `ArrayList`. The `MappingIterator` enables this program to access each record in the CSV file. **`CsvMapper`'s `readerFor` method** receives `Account.class` to indicate it will create `Account` objects. The **`with` method** provides the schema Jackson will use when deserializing the CSV records into `Account` objects. Finally, the **`readValues` method** indicates that the `MappingIterator` will read data from the file specified in its argument.
- Line 35 calls the **`MappingIterator`'s `readAll` method** to iterate through all the records in the CSV file, reading them into a `List<Account>`. This makes it easy to work with all the CSV file's data in memory.

Caution: Commas in CSV Data Fields

Be careful when working with strings containing embedded commas, such as the name `"Flores, Silvio"`. If this name were accidentally stored as the two strings `"Flores"` and `"Silvio"`, that CSV record would have four fields, not three. Programs that read CSV files typically expect every record to have the same number of fields; otherwise, problems occur.

Caution: Missing Commas and Extra Commas in CSV Files

Be careful when preparing and processing CSV files. For example, suppose your file is composed of records, each with four comma-separated `int` values, such as

```
100,85,77,9
```

If you accidentally omit one of these commas, as in

```
100,8577,9
```

then the record has only three fields, one with the invalid value `8577`.

If you put two adjacent commas where only one is expected, as in

```
100,85,,77,9
```

then you have five fields rather than four, and one of the fields erroneously would be empty. Each of these comma-related errors could confuse programs trying to process the record.

Generative AI

1 Prompt genAIs, asking for an example of using Jackson to work with CSV fields that contain commas.
2 Prompt genAIs for best practices when working with CSV files.

11.8 Case Study: Reading and Analyzing a CSV File Containing the *Titanic* Disaster Dataset

The ***Titanic* disaster dataset**[11] is one of the most commonly used datasets for data analytics beginners. It lists all the passengers and whether they survived when the ship *Titanic* struck an iceberg and sank during its first voyage of April 10–15, 1912. In this case, study, we'll load the dataset, view some of its data and perform some basic data analytics.

11. "TitanicSurvival" dataset on `https://vincentarelbundock.github.io/Rdatasets`. Dataset authors: Frank E. Harrell, Jr., and Thomas Cason.

The dataset is available from numerous dataset repositories. We used the version from the Rdatasets repository at

Click here to view code image

```
https://vincentarelbundock.github.io/Rdatasets/csv/carData/
   TitanicSurvival.csv
```

We placed this file in the same folder as the example's code.

Getting to Know Your Data

In data analytics, you'll often begin by getting to know your data. One way is to simply look at the raw data. If you open the `TitanicSurvival.csv` file in a text editor or spreadsheet application, you'll see that the dataset contains 1,309 rows, each with five columns—often called **features** in data analytics. We ignore the first column containing the passenger's names and use only the last four columns in this case study:

- `survived`: `"yes"` or `"no"`.
- `sex`: `"female"` or `"male"`.
- `age`: The passenger's age. Most ages are integers, but some children under 1 year of age have floating-point values, so we'll process this column as `double` values.
- `passengerClass`: `"1st"`, `"2nd"` or `"3rd"` for first, second or third class, respectively.

To learn more about the dataset's origins, visit

Click here to view code image

```
https://hbiostat.org/data/repo/titanic
```

Missing Data

Bad data values and missing values can significantly impact data analysis. The *Titanic* dataset is missing the `age`s for 263 passengers. These are represented as empty fields in the CSV file—that is, there is no data between the commas that delimit the age field in those records. In this example, when we produce descriptive statistics for the passengers' ages, we'll "clean" the data to filter out and ignore the missing values. In data science, preparing data for analysis is called **data munging** or **data wrangling**.

Some data scientists advise against any attempts to insert "reasonable values." Instead, they advocate clearly marking missing data and leaving it up to a data analytics package to handle the issue. Others offer strong cautions.[12]

12. This footnote was abstracted from a comment sent to us July 20, 2018, by one of our Python textbook's academic reviewers, Dr. Alison Sanchez of the University of San Diego School of Business. She commented: "Be cautious when mentioning 'substituting reasonable values' for missing or bad values. A stern warning: 'Substituting' values that increase statistical significance or give more 'reasonable' or 'better' results is not permitted. 'Substituting' data should not turn into

‘fudging’ data. The first rule students should learn is not to eliminate or change values that contradict their hypotheses. ‘Substituting reasonable values‘ does not mean students should feel free to change values to get the results they want.”

Loading the Dataset

Figure 11.10 loads the dataset and performs our analysis. We broke the program into parts for discussion purposes. Remember to add Jackson to the classpath when compiling and running this program.

Click here to view code image

```
1  // Fig. 11.10: TitanicAnalysis.java
2  // Reading the Titanic dataset from a CSV file, then analyzing it.
3  import com.fasterxml.jackson.annotation.JsonIgnoreProperties;
4  import com.fasterxml.jackson.databind.MappingIterator;
5  import com.fasterxml.jackson.dataformat.csv.CsvMapper;
6  import com.fasterxml.jackson.dataformat.csv.CsvSchema;
7  import java.nio.file.Path;
8  import java.util.ArrayList;
9  import java.util.List;
10
```

Fig. 11.10 | Reading the Titanic dataset from a CSV file, then analyzing it.

`record` Class `TitanicAnalysis` for Storing Each Dataset Record

Jackson can read CSV records and create objects of a type you specify so you can conveniently access the record’s fields. Lines 13–19 declare the `record` class `TitanicRecord`, which has instance variables representing the four dataset columns we use in this example. As discussed previously, line 13 tells Jackson to ignore JSON name-value pairs that do not have a matching instance variable in class `TitanicRecord`.

Click here to view code image

```
11 public class TitanicAnalysis {
12    // record class representing the fields in each dataset row
13    @JsonIgnoreProperties(ignoreUnknown = true)
14    public record TitanicRecord(
15       String survived,
16       String sex,
17       String age,
```

```
18          String passengerClass
19      ) {}
20
```

Loading the Dataset

Lines 22 and 26 create the `CsvMapper` and `CsvSchema` we'll use to configure Jackson to read the CSV file and convert each record to a `TitanicRecord` object. In this example, we use **`CsvSchema`'s `static` `emptySchema` method** to indicate that Jackson should automatically determine how to map the CSV records to `TitanicRecord`s. At the end of this section, you'll prompt genAIs, asking for the benefits and disadvantages of this approach. Lines 30–32 configure the `MappingIterator` that converts each record to a `TitanicRecord` object, then line 35 uses it to read all the CSV file's records into a `List<TitanicRecord>`.

Click here to view code image

```
21      public static void main(String[] args) throws Exception {
22          var mapper = new CsvMapper(); // reads CSV records
23
24          // specify that Jackson should map each CSV column
25          // to an object based on the column names
26          CsvSchema schema = CsvSchema.emptySchema().withHeader();
27
28          // create the MappingIterator that knows how to read
29          // the file's records into TitanicRecord objects
30          MappingIterator<TitanicRecord> iterator =
31             mapper.readerFor(TitanicRecord.class).with(schema).readValues(
32                Path.of("TitanicSurvival.csv").toFile());
33
34          // load dataset into a List of TitanicRecord objects
35          List<TitanicRecord> titanic = iterator.readAll();
36
```

Extracting the Column Data into `ArrayList`s

Lines 38–41 create `ArrayList`s for each column. The loop at lines 44–58 populates each `ArrayList` with the corresponding column's data. As we add each age to the `age` `Array-List`, we check whether it's empty. If not, we convert it to a `double` and add it to the `ArrayList`; otherwise, we add the value `Double.NaN` (not a number) to indicate the missing value. You'll learn in Chapter 12 that Java collections can store only objects and that `double`s and `int`s are converted to their `Double` object and `Integer` object equivalents

(and vice versa) automatically. As we add each passenger class to the `pclass ArrayList`, we convert it from the `String "1st"`, `"2nd"` or `"3rd"` to the number `1`, `2` or `3`. We'll use these as array indexes when summarizing the number of passengers in each class.

Click here to view code image

```
37          // create lists representing each column with the correct type
38          var survived = new ArrayList<String>();
39          var sex = new ArrayList<String>();
40          var age = new ArrayList<Double>();
41          var pclass = new ArrayList<Integer>();
42
43          // populate the column lists
44          for (TitanicRecord record : titanic) {
45             survived.add(record.survived());
46             sex.add(record.sex());
47             age.add(!record.age().isEmpty() ?
48                Double.parseDouble(record.age()) : Double.NaN);
49             pclass.add(
50                switch (record.passengerClass()) {
51                   case "1st" -> 1;
52                   case "2nd" -> 2;
53                   case "3rd" -> 3;
54                   default -> throw new IllegalArgumentException(
55                      "Unknown class: " + record.passengerClass());
56                }
57             );
58          }
59
```

Viewing Some Rows in the *Titanic* Dataset

The 1,309 rows each represent one passenger. According to Wikipedia, there were approximately 1,317 passengers, and 817 of them died.[13] For large datasets, it's possible to display only small portions of the data. A common practice when getting to know your data is to display a few rows from the beginning and end of the dataset to get a sense of the data. The code in lines 64–67 displays the first five elements of each column's data:

13. "Number of people aboard Titanic and number of lost and saved." Accessed January 1, 2025. https://www.titanicpages.com/numberaboardandlostandsaved.

Click here to view code image

```
60          // display first 5 rows
```

```
61          System.out.printf("First five rows:%n%-10s%-8s%-6s%s%n",
62             "survived", "sex", "age", "class");
63
64          for (int i = 0; i < 5; ++i) {
65             System.out.printf("%-10s%-8s%-6.1f%d%n",
66                survived.get(i), sex.get(i), age.get(i), pclass.get(i));
67          }
68
```

```
First five rows:
survived  sex     age   class
yes       female  29.0  1
yes       male    0.9   1
no        female  2.0   1
no        male    30.0  1
no        female  25.0  1
```

The code in lines 70–77 displays the last five elements of each column's data. Note the value `NaN`, indicating a missing value for one of the `age` column's rows.

Click here to view code image

```
69          // display last 5 rows
70          System.out.printf("%nLast five rows:%n%-10s%-8s%-6s%s%n",
71             "survived", "sex", "age", "class");
72          int count = titanic.size();
73
74          for (int i = count - 5; i < count; ++i) {
75             System.out.printf("%-10s%-8s%-6.1f%d%n",
76                survived.get(i), sex.get(i), age.get(i), pclass.get(i));
77          }
78
```

```
Last five rows:
survived  sex     age   class
no        female  14.5  3
no        female  NaN   3
no        male    26.5  3
no        male    27.0  3
no        male    29.0  3
```

Data Munging: Cleaning the Age Column for Calculating Descriptive Statistics

As part of getting to know a dataset, data scientists often use

statistics to describe and summarize data. We'll calculate several **descriptive statistics** for the age column, including the number of passengers for which we have ages, and the average, minimum, maximum and median age values. Before performing these calculations, we must remove `NaN` values, because a calculation that includes the value `NaN` produces `NaN` as the result. So, lines 80–85 create the `ArrayList cleanAge` containing only the values from `age` that are not `NaN`. **`Double` method `isNaN`** returns `true` if the value is `NaN`.

Click here to view code image

```
79          // clean the age column (remove NaN)
80          var cleanAge = new ArrayList<Double>();
81          for (Double value : age) {
82             if (!value.isNaN()) {
83                cleanAge.add(value);
84             }
85          }
86
```

Basic Descriptive Statistics for the Cleaned Age Column

Now, we can calculate the descriptive statistics. Line 88 sorts `cleanAge`, so we can determine the minimum, maximum and median values. When `sort`'s argument is `null` the `ArrayList`'s element type must implement the `Comparable` interface so `sort` can compare the elements to determine sort order—class `Double` implements `Comparable`. To count the number of people for which we have valid `age`s, we simply get `cleanAge`'s `size` (line 89). Lines 92–94 determine the median. If `cleanAge`'s `size` is even, the median is the average of the two middle elements (line 93); otherwise, it's the middle element (line 94). Lines 97–103 calculate the average age.

Click here to view code image

```
87          // descriptive statistics for cleaned age column
88          cleanAge.sort(null); // helps determine min, max and median ages
89          int size = cleanAge.size();
90
91          // get median age
92          double medianAge = (size % 2 == 0) ?
93             (cleanAge.get(size / 2 - 1) + cleanAge.get(size / 2)) / 2.0 :
94             cleanAge.get(size / 2);
```

```
95
96          // calculate average of ages
97          double sum = 0.0;
98
99          for (double value : cleanAge) {
100             sum += value;
101         }
102
103         double averageAge = sum / size;
104
```

Lines 106–111 display the descriptive statistics. The `ArrayList` is sorted, so lines 109–110 call the **`ArrayList`'s `getFirst` and `getLast` methods** to get first and last elements, which contain the minimum and maximum values, respectively. The average and median are **measures of central tendency**. Each produces a single value that represents a "central" value in a set of values—a value that is, in some sense, typical of the others. For the `1046` people with valid `age`s, the average `age` was `29.88` years old. The youngest passenger (i.e., the minimum) was just over two months old (`0.17 * 12` is `2.04`), and the oldest (i.e., the maximum) was 80. The `median age` was `28`.

Click here to view code image

```
105         // display the age statistics
106         System.out.printf("%nDescriptive statistics for the age column:%n");
107         System.out.printf("Passengers with age data: %d%n", size);
108         System.out.printf("Average age: %.2f%n", averageAge);
109         System.out.printf("Minimum age: %.2f%n", cleanAge.getFirst());
110         System.out.printf("Maximum age: %.2f%n", cleanAge.getLast());
111         System.out.printf("Median age: %.2f%n", medianAge);
112
```

```
Descriptive statistics for the age column:
Passengers with age data: 1046
Average age: 29.88
Minimum age: 0.17
Maximum age: 80.00
Median age: 28.00
```

Determining Passenger Counts By Class

Let's calculate each class's number of passengers. Lines 114–118 count the number of passengers in each class—we ignore element 0 of the `counts` array so we can use each class's numeric value as an

index. Lines 120–122 display the counts.

Click here to view code image

```
113         // passenger counts by class
114         int[] counts = new int[4]; // ignore element 0
115
116         for (int value : pclass) {
117            ++counts[value];
118         }
119
120         System.out.printf(
121            "%nPassenger counts by class:%n1st: %d%n2nd: %d%n3rd: %d%n",
122            counts[1], counts[2], counts[3]);
123
```

```
Passenger counts by class:
1st: 323
2nd: 277
3rd: 709
```

Counting By Sex and By Passenger Class the Numbers of People Who Survived

Recall that the `survived` column contains `"yes"` or `"no"` to represent survived or died, respectively. Lines 129–142 iterate through the `survived ArrayList`, testing whether each value is `"yes"`. If so, we increment the `survivorCount`. Next, lines 133–138 determine whether the survivor was a `"female"` or `"male"` and update the appropriate counter in the `survivorsBySex` array. Then, line 140 increments the appropriate counter in `survivorsBy-Class` —again, we ignore element `0`. We'll use these counts to calculate percentages.

Click here to view code image

```
124         // summarize counts of people who survived by sex and by class
125         int survivorCount = 0;
126         int[] survivorsBySex = new int[2];
127         int[] survivorsByClass = new int[4]; // ignore element 0
128
129         for (int i = 0; i < survived.size(); ++i) {
130            if (survived.get(i).equals("yes")) {
131               ++survivorCount;
132
133               if (sex.get(i).equals("female")) {
134                  ++survivorsBySex[0];
```

```
135             }
136             else {
137                ++survivorsBySex[1];
138             }
139
140             ++survivorsByClass[pclass.get(i)];
141          }
142       }
143
```

Calculating and Displaying the Percentages of People Who Survived

Lines 145–146 display the number of people who survived and the number who died. To determine how many people died, line 146 subtracts `survivorCount` from the `survived ArrayList`'s size. Lines 147–148 calculate and display the percentage of people who survived. Lines 149–158 calculate and display the percentages of:

- females who survived,
- males who survived,
- first-class passengers who survived,
- second-class passengers who survived, and
- third-class passengers who survived.

Of the survivors, about two-thirds were female, and first-class passengers survived at a higher rate than the other passenger classes.

Click here to view code image

```
144       // calculate and display the survivor statistics
145       System.out.printf("%nSurvived count: %d%nDied count: %d%n",
146          survivorCount, survived.size() - survivorCount);
147       System.out.printf("Percent who survived: %.2f%%%n",
148          100.0 * survivorCount / survived.size());
149       System.out.printf("Female survivor percentage: %.2f%%%n",
150          100.0 * survivorsBySex[0] / survivorCount);
151       System.out.printf("Male survivor percentage: %.2f%%%n",
152          100.0 * survivorsBySex[1] / survivorCount);
153       System.out.printf("1st class survivor percentage: %.2f%%%n",
154          100.0 * survivorsByClass[1] / survivorCount);
155       System.out.printf("2nd class survivor percentage: %.2f%%%n",
156          100.0 * survivorsByClass[2] / survivorCount);
157       System.out.printf("3rd class survivor percentage: %.2f%%%n",
```

```
            100.0 * survivorsByClass[3] / survivorCount);
    }
}
```

```
Survived count: 500
Died count: 809
Percent who survived: 38.20%
Female survivor percentage: 67.80%
Male survivor percentage: 32.20%
1st class survivor percentage: 40.00%
2nd class survivor percentage: 23.80%
3rd class survivor percentage: 36.20%
```

Generative AI

1 Prompt genAIs with Fig. 11.10's code, asking them to make the `TitanicRecord` class's age instance variable a `double` and to use a custom deserializer to replace missing values from that column with `Double.NaN`.
2 Prompt genAIs with the code from Figs. 11.9 and 11.10, asking them to compare the approaches to creating the `CsvSchema` in each. Also, ask them to explain the benefits and disadvantages of the two approaches.

11.9 Objects-Natural Security Case Study: RSA Public-Key Cryptography

In Section 7.9, you began learning about secret-key cryptography. The sender's plaintext was encrypted with a secret key to form ciphertext. The receiver used the same secret key to decode the ciphertext, forming the original plaintext—this process is called symmetric encryption because the same key is used by both the sender and the recipient.

Public-Key Cryptography

A problem with secret-key cryptography is that the security of the ciphertext is only as good as the security of the secret key, and several copies of that key are "floating around." In their paper "New

Directions in Cryptography,"[14] Whitfield Diffie and Martin Hellman, introduced **public-key cryptography** to address the weakness of secret-key cryptography—the vulnerability of the secret key having to be known by both the sender and the recipient. They came up with the idea but not an implementation of the scheme.

14. "New Directions in Cryptography." November 6, 1976. Accessed January 1, 2025. https://ee.stanford.edu/~hellman/publications/24.pdf.

RSA Public-Key Cryptography

Rivest, Shamir and Adelman were the first to publish a working implementation of public-key cryptography. The scheme, called RSA, bears the initials of their last names.[15] RSA is one of the world's most widely implemented public-key cryptography schemes.[16] Because RSA can be slow,[17] many organizations prefer to stick to faster private-key encryption, using RSA only to securely send the secret key. Such encryption and decryption techniques are critical for securing data stored in files and transmitted over networks.

15. R. Rivest; A. Shamir; L. Adleman (February 1978). "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems" (PDF). Communications of the ACM. 21 (2): 120–126. Accessed January 1, 2025. https://people.csail.mit.edu/rivest/Rsapaper.pdf.
16. Cobb, Michael. "RSA algorithm (Rivest-Shamir-Adleman)." Accessed January 1, 2025. https://searchsecurity.techtarget.com/definition/RSA.
17. "RSA (cryptosystem)." Accessed January 1, 2025. https://en.wikipedia.org/wiki/RSA_(cryptosystem).

Historical Note

Clifford Cocks in the U.K. created a workable public-key scheme several years before the RSA paper was published,[18] but his work was classified, so it was not revealed until about 20 years after RSA appeared. The company RSA Security held a patent on the RSA algorithm. In 2000, that patent was coming due for renewal—instead of renewing, they placed the algorithm into the public domain.[19]

18. "Clifford Cocks." Accessed January 1, 2025. https://en.wikipedia.org/wiki/Clifford_Cocks.
19. "RSA Security Releases RSA Encryption Algorithm into Public

Domain." September 6, 2000. Accessed January 1, 2025. https://web.archive.org/web/20071120112201/http://www.rsa.com/press_release.aspx?id=261.

Enhanced Cryptography Demo

In this case study, we enhance our AES private-key encryption example to demonstrate how to securely encrypt the secret key using **RSA public-key cryptography**.[20,21,22] In particular, we focus on how to generate:

- the **public key** any sender can use to encrypt an AES secret key for transmission to a specific recipient, and
- the **private key** only the specific recipient can use to decrypt the AES secret key.

20. "RSA (cryptosystem)." Accessed January 1, 2025. https://en.wikipedia.org/wiki/RSA_(cryptosystem).
21. "RSA Algorithm." Accessed January 1, 2025. https://simple.wikipedia.org/wiki/RSA_algorithm.
22. "PKCS #1: RSA Cryptography Specifications Version 2.2." November 2016. Accessed January 1, 2025. https://tools.ietf.org/html/rfc8017.

Public-key cryptography is also called asymmetric-key cryptography because different keys are used for encryption and decryption. This technique is often used to transmit the secret keys used in symmetric encryption. RSA is based on sophisticated mathematics using enormous prime numbers consisting of hundreds of digits, but the Java cryptography APIs encapsulate this complexity, hiding it from you. Much of Fig. 11.11's code is identical to the AES cryptography demo in Fig. 7.23, so we focus on the enhancements. We split the example into parts for discussion purposes.

Click here to view code image

```
1  // Fig. 11.11: CryptographyDemo.java
2  // Encrypting and decrypting messages with AES,
3  // and using RSA to encrypt and decrypt the AES key.
4  import javax.crypto.Cipher;
5  import javax.crypto.KeyGenerator;
6  import javax.crypto.SecretKey;
7  import javax.crypto.spec.SecretKeySpec;
8  import java.security.KeyPair;
9  import java.security.KeyPairGenerator;
```

```
10  import java.security.PublicKey;
11  import java.security.PrivateKey;
12  import java.security.SecureRandom;
13  import java.util.Base64;
14  import java.util.Scanner;
15
16  public class CryptographyDemo {
17     public static void main(String[] args) {
18        var scanner = new Scanner(System.in);
19
```

Fig. 11.11 | Encrypting and decrypting messages with AES, and using RSA to encrypt and decrypt the AES key.

Getting the Message to Encrypt and AES Encrypting It

Lines 20–35 perform the same tasks as in Section 7.9:

- Input the plain text to encrypt,
- input the secret key seed for reproducibility (again, for demo purposes only),
- generate the secret key for AES encryption, and
- use the secret key to AES encrypt the plain text.

Click here to view code image

```
20        System.out.println("Enter the text to encrypt:");
21        String plaintext = scanner.nextLine();
22
23        System.out.println(
24           "\nEnter the secret key seed (for reproducibility):");
25        String seedString = scanner.nextLine();
26
27        try { // encryption/decryption might throw exceptions
28           // generate a reproducible AES key from the seedString
29           SecretKey secretKey = generateKey(seedString);
30           System.out.printf("%nSecretKey:%n%s%n",
31              Base64.getEncoder().encodeToString(secretKey.getEncoded()));
32
33           // AES encrypt the plaintext
34           String ciphertext = encrypt(plaintext, secretKey);
35           System.out.printf("%nEncrypted message:%n%s%n", ciphertext);
36
```

```
Enter the text to encrypt:
```

```
Welcome to encryption and decryption using AES and RSA in Java!

Enter the secret key seed (for reproducibility):
XMWUJBVYHXZ

SecretKey:
8ew8jBJVSpkRtG4zorwxd34JBpUxEdOt+ec0QTfr7cE=

Encrypted message:
Cz9o3YoFcX+yebfAOuMPMi+UZz2bc5jJdstfzapaNKyvWTHQf2nmPHQ9SjRVue1Qc179qM+
wGI44tpOHKjErKrA==
```

Generating the RSA Public and Private Keys

Lines 38–45 generate an RSA key pair that we'll use to simulate the secure transmission of an encrypted AES secret key:

- Line 38 calls the **`KeyPairGenerator` `static` method** **`getInstance`** with the argument `"RSA"` to get a `KeyPairGenerator` object for RSA encryption.
- Line 39 uses the **`KeyPairGenerator` method** **`initialize`** to set the key size to 2048 bits, as the National Institute of Standards and Technology (NIST) recommends[23]—this defines the encryption strength.

  23. National Institute of Standards and Technology (NIST). Recommendation for Key Management: Part 3 — Application-Specific Key Management Guidance. Special Publication 800-57 Part 3 Revision 1. Gaithersburg, MD: National Institute of Standards and Technology, 2015. https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-57Pt3r1.pdf.

- Line 40 generates the RSA **`KeyPair`** by calling the **`KeyPairGenerator` method** **`generateKeyPair`**.
- Lines 44–45 get the `KeyPair`'s `PublicKey` and a `PrivateKey` via **`KeyPair` methods** **`getPublic`** and **`getPrivate`**.

You share your `PublicKey` with anyone sending you encrypted data. You keep the `Private-Key` secret and use it to decrypt data encrypted with your `PublicKey`, ensuring that only the intended recipient (you) can read the message.

Click here to view code image

```
37          // generate RSA key pair -- normally done by message receiver
38          var keyPairGenerator = KeyPairGenerator.getInstance("RSA");
39          keyPairGenerator.initialize(2048); // set key size to 2048 bits
```

```
40          KeyPair rsaKeyPair = keyPairGenerator.generateKeyPair();
41
42          // get RSA public and private keys -- recipient sends the
43          // public key to anyone who needs to send them something securely
44          PublicKey publicKey = rsaKeyPair.getPublic();
45          PrivateKey privateKey = rsaKeyPair.getPrivate();
46
```

Encrypting the AES Secret Key

Lines 49–50 use RSA to encrypt the AES `SecretKey`. Method `encryptRSA` (lines 117–130) takes the `SecretKey`'s `byte` array representation (`secretKey.getEncoded()`) and the RSA `publicKey` and uses the `publicKey` to encrypt the AES `SecretKey`—this ensures only the intended recipient can later decrypt it using their corresponding RSA private key. Encrypting the AES secret key allows it to be transmitted securely. For demo purposes, we display the encrypted AES `SecretKey` so you can confirm later that it was decrypted properly—your output will differ as the RSA `publicKey` was generated randomly. In real-world cryptography, we'd send the `cipherText` (line 34) and the `encryptedAESKey` to the recipient.

Click here to view code image

```
47          // encrypt the AES secret key with the RSA public key;
48          // this would be sent to recipient with the encrypted message
49          String encryptedAESKey =
50             encryptRSA(secretKey.getEncoded(), publicKey);
51          System.out.printf(
52             "%nRSA Encrypted AES SecretKey:%n%s%n", encryptedAESKey);
53
```

```
RSA Encrypted AES SecretKey:
RzZ2APUmXjwjWnKiCptMifxdwgADNs9k3qXEbUtTivXfD/kaUd3rExhgfTiBqnbIYdTuuze3C5xxy
l66AEf3Hu7bP0rJbPjbp3yhOD6e/x5Gf//UA+b4d1GM+dOpMAX0F60iu658W33M+PTCowDcOyj9xE
8yYsdtW3E4/uJ2CQMt22+87LbbtDwFVqT8x91ZWuqg+gKs7RhiybdZ7z8JzjmEI+Wr5UH0AXT7qvj
SQjZGTKt0uZfJniwPT4isJWHtPPwX1UKwk7GtG5vlnOWgopYu09bmee2xBVMc/LUzbOpeEIs1J435
bw1bYoaeJiIjVyiyWyjcvsW6btJPNHXZeg==
```

Decrypting the AES Secret Key

To decrypt the `ciphertext`, the recipient must first decrypt the AES secret key that was used to produce the `ciphertext`. Lines 55–56 call method `decryptRSA` (lines 133–148) to decrypt the AES `SecretKey` using RSA decryption. The method `decryptRSA` receives the encrypted AES key and the RSA private key—the only key that

can decrypt data encrypted by the corresponding public key, ensuring the AES secret key's security. This process ensures that only the intended recipient, who possesses the private key, can obtain the original AES key and read the encrypted message. For demonstration purposes, we display the decrypted AES secret key, so you can compare it to the one output by lines 30–31 and see that it was decrypted properly.

Click here to view code image

```
54            // receiver decrypts AES secret key using RSA private key
55            SecretKey decryptedSecretKey =
56               decryptRSA(encryptedAESKey, privateKey);
57            System.out.printf("%nDecrypted AES SecretKey:%n%s%n",
58               Base64.getEncoder().encodeToString(
59                  decryptedSecretKey.getEncoded()));
60
```

```
Decrypted AES SecretKey:
8ew8jBJVSpkRtG4zorwxd34JBpUxEdOt+ec0QTfr7cE=
```

Decrypting the AES-Encrypted Ciphertext

Lines 62–63 perform the same tasks as Section 7.9 to decrypt the AES encrypted message but use the decrypted secret key from lines 55–56. In a real-world scenario, this would be done by the message's receiver.

Click here to view code image

```
61            // AES decrypt the text
62            System.out.printf("%nDecrypted:%n%s%n",
63               decrypt(ciphertext, decryptedSecretKey));
64         }
65         catch (Exception ex) {
66            System.out.printf("""
67               An exception occurred during encryption/decryption:
68               %s%n
69               """, ex);
70         }
71      }
72
```

```
Decrypted:
Welcome to encryption and decryption using AES and RSA in Java!
```

`generateKey`, `encrypt` and `decrypt` Methods

The methods in lines 76–113 perform the same tasks as in Section 7.9 for generating an AES secret key, encrypting plain text using an AES secret key and decrypting ciphertext using the same AES secret key.

Click here to view code image

```
73      // Generate and return a SecretKey. Note: We are seeding a
74      // SecureRandom object for reproducibility so you get the same results
75      // we show in this program's output. Don't do this in production code.
76      public static SecretKey generateKey(String seed) throws Exception {
77         // get a secure random number generator
78         var random = SecureRandom.getInstance("SHA1PRNG");
79         random.setSeed(seed.getBytes()); // for demo reproducibility
80
81         // get an AES SecretKey generator
82         var keyGen = KeyGenerator.getInstance("AES");
83         keyGen.init(256, random); // set up for 256-bit keys
84         return keyGen.generateKey(); // generate and return new SecretKey
85      }
86
87      // encrypt a String using AES
88      public static String encrypt(
89         String plaintext, SecretKey secretKey) throws Exception {
90
91         var cipher = Cipher.getInstance("AES"); // get a Cipher object
92         cipher.init(Cipher.ENCRYPT_MODE, secretKey); // configure to encrypt
93
94         // encrypt the byte[] representation of plaintext
95         byte[] encrypted = cipher.doFinal(plaintext.getBytes());
96
97         // Base64 encode encrypted as a ciphertext String and return
98         return Base64.getEncoder().encodeToString(encrypted);
99      }
100
101     // decrypt an AES-encrypted String
102     public static String decrypt(
103        String ciphertext, SecretKey secretKey) throws Exception {
104
105        var cipher = Cipher.getInstance("AES"); // get a Cipher object
106        cipher.init(Cipher.DECRYPT_MODE, secretKey); // configure to decrypt
107
108        // Base64 decode ciphertext to byte array
109        byte[] ciphertextBytes = Base64.getDecoder().decode(ciphertext);
110
111        // create plaintext String from ciphertextBytes and return it
112        return new String(cipher.doFinal(ciphertextBytes));
113     }
```

Encrypting the AES Secret Key Using RSA

Method `encryptRSA` (lines 117–130) encrypts a `byte` array using an RSA `PublicKey`:

- Line 120 creates a `Cipher` object configured to use the RSA algorithm.
- Line 123 initializes the `Cipher` object for encryption mode using the RSA public key. This ensures that only the recipient with the corresponding private key can decrypt the data.
- Line 126 calls the `Cipher` object's `doFinal` to encrypt the `byte` array `data`, which, in this case, is the AES secret key.
- Finally, line 129 encodes the encrypted data as a `Base64 String` and returns a `String` representing the encrypted AES secret key.

This encrypted key can then be safely sent to the recipient along with the encrypted message, so they can use their RSA private key to decrypt the AES secret key and decrypt the message.

Click here to view code image

```
115     // encrypt data using RSA public key -- the result of this method
116     // would be sent to the AES encrypted message's receiver
117     public static String encryptRSA(byte[] data, PublicKey publicKey)
118        throws Exception {
119
120        var cipher = Cipher.getInstance("RSA"); // create RSA Cipher object
121
122        // initialize cipher in encryption mode with recipient's public key
123        cipher.init(Cipher.ENCRYPT_MODE, publicKey);
124
125        // encrypt data (the AES key) using the RSA public key
126        byte[] encryptedData = cipher.doFinal(data);
127
128        // Base64 encode the encrypted AES key
129        return Base64.getEncoder().encodeToString(encryptedData);
130     }
131
```

Decrypting the AES Secret Key Using RSA

Method `decryptRSA` (lines 133–148) decrypts data using an RSA

`PrivateKey` and returns the decrypted data as an AES `SecretKey`:

- Line 136 creates a `Cipher` object configured for the RSA algorithm.
- Line 139 initializes the `Cipher` object in decryption mode with the recipient's `PrivateKey`—only this key can decrypt the data that was encrypted with the corresponding `PublicKey`.
- Lines 142–143 decode the encrypted data from its `Base64 String` format to a `byte` array, then call the `Cipher` object's `doFinal` to decrypt the array, resulting in the original AES secret key as a `byte` array.
- Finally, the method converts this `byte` array to an AES `SecretKey` object, which the recipient can use to decrypt of the corresponding message. The **`Secret-KeySpec` class** is an implementation of the `SecretKey` interface.

This process ensures that only the intended recipient, who has the RSA `PrivateKey`, can decrypt and use the AES `SecretKey`.

Click here to view code image

```
132     // decrypt data using RSA private key and return the SecretKey
133     public static SecretKey decryptRSA(
134        String encryptedData, PrivateKey privateKey) throws Exception {
135
136        var cipher = Cipher.getInstance("RSA"); // create RSA Cipher object
137
138        // initialize cipher in decryption mode with recipient's private key
139        cipher.init(Cipher.DECRYPT_MODE, privateKey);
140
141        // decode the encrypted AES key from Base64 format and decrypt it
142        byte[] decryptedData =
143           cipher.doFinal(Base64.getDecoder().decode(encryptedData));
144
145        // convert decrypted byte array back to an AES SecretKey object
146        return new SecretKeySpec(
147           decryptedData, 0, decryptedData.length, "AES");
148     }
149  }
```

Generative AI

1 Prompt genAIs, asking why "symmetric-key cryptography" is a

good synonym for private-key cryptography.
2 Prompt genAIs, asking why "asymmetric-key cryptography" is a good synonym for public-key cryptography.

11.10 Wrap-Up

In this chapter, you created and manipulated persistent data on secondary storage devices. We began with Java's file-handling architecture, explained that data can be stored in text or binary files, and discussed their differences. We demonstrated Java NIO classes and interfaces for retrieving information about files and directories. We discussed using `Formatter` objects to write to and `Scanner` objects to read from text files. We created and manipulated sequential text files.

Next, we presented several case studies on transforming objects to and from popular data formats. The first introduced the human-and-computer-readable JavaScript Object Notation (JSON) data format. We used the popular Java-based Jackson open-source library to serialize objects to a file in JSON format, then deserialized the JSON text from the file into objects in memory. The second JSON case study used Java's `HttpClient`, `HttpRequest` and `HttpResponse` classes to invoke a free `OpenWeatherMap.org` web service that returned a city's weather report in JSON format. Then, we used Jackson to process the response and display the weather report.

The next two case studies introduced the popular CSV (comma-separated-values) file format and how to write and read CSV files using the Jackson open-source library. Then, we used CSV reading capabilities to load the Titanic disaster dataset, which is commonly used to introduce basic data analytics.

Finally, we presented our last objects-natural case study. We enhanced Section 7.9's private-key cryptography example by using RSA public-key cryptography to encrypt and decrypt the AES secret key, enabling secure transmission of the secret key to an encrypted message's recipient. Such encryption and decryption techniques are critical for securing data stored in files and transmitted over networks. We'll continue our discussions of persisting data in Chapter 20, Accessing Databases with JDBC and SQLite.

In the next chapter, we discuss the Java Collections Framework, which contains many predefined collection classes and interfaces for manipulating data in memory, like the `ArrayList` class

(introduced in Section 6.20) and the `List` interface (introduced in Section 11.4.1).

Special Section: Generative AI Exercises

For each of the following questions, try multiple genAIs to determine which performs best. Always verify the results. Keep in mind that they sometimes make mistakes and even hallucinate, creating unreliable information.

11.1 ***(Files Class Tutorial)*** Prompt genAIs to write a brief tutorial with code examples on the Java `Files` class from package `java.nio.files`.

11.2 ***(PrintWriter vs. Formatter for Formatted Output)*** Prompt genAIs, asking them to explain the key differences between `java.io.PrintWriter` and `java.util.Formatter` and when you should use each.

11.3 ***(Telephone-Number Word Generator)*** Standard telephone keypads contain the digits zero through nine. The numbers 2–6 and 8 each have three letters associated with them and 7 and 9 each have four, as shown in the following table:

Digits	Letters
2	A B C
3	D E F
4	G H I
5	J K L
6	M N O
7	P Q R S
8	T U V
9	W X Y Z

Many people find it challenging to memorize phone numbers, so they use the correspondence between digits and letters to develop seven-letter words corresponding to their phone numbers. For example, a person whose telephone number is 686-2377 might use the correspondence in the preceding table to develop the seven-letter word "NUMBERS." Every seven-letter word corresponds to one seven-digit telephone number. A restaurant wishing to increase its takeout business could surely do so with the number 825-3688 (i.e., "TAKEOUT").

Every seven-letter phone number corresponds to many different seven-letter words, but most represent unrecognizable

juxtapositions of letters. It's possible, however, that a barbershop owner would be pleased to know that the shop's telephone number, 424-7288, corresponds to "HAIRCUT." A veterinarian with the phone number 738-2273 would be pleased to know that the number corresponds to "PETCARE." An automotive dealership would be happy to know that the dealership number, 639-2277, corresponds to "NEWCARS."

Prompt genAIs to write a Java program that, given a seven-digit number, uses a `Formatter` object to write to a file every possible seven-letter word combination corresponding to that number. Next, upload the program's output file to genAIs, asking them to analyze the output and select only the real words or groups of words while automatically eliminating profanity. For groups of words (e.g., "PETCARE") ask the genAIs to display them with spaces between the words (e.g., "PET CARE"). In **Chapter 19, Building API-Based Java Generative AI Applications**, we'll discuss OpenAI's Moderation APIs, which enable you to filter content containing harmful text and images.

11.4 ***(Telephone-Number Word Generator)*** Use genAI to extend Exercise 11.3 to work with 10-digit phone numbers.

11.5 ***(Tablesaw Library)*** Prompt genAIs, asking for an overview of the Tablesaw open-source library (`https://github.com/jtablesaw/tablesaw`) and its data analytics capabilities. Next, prompt the genAIs with Section 11.8's Titanic disaster dataset example, asking them to rewrite the code using the Tablesaw library and explain their code line by line. Also, ask them to provide precise instructions on compiling and running the program using `javac` and `java` (or your preferred IDE).

11.6 ***(Binary File Processing)*** Prompt genAIs to write and explain a few simple code examples demonstrating Java binary file processing.

11.7 ***(Storing RSA Private Keys Securely)*** Prompt genAIs, asking how to securely store your RSA private key using Java.

11.8 ***(Best Practices for Processing Large Files)*** Prompt genAIs for a list of best practices for processing large files efficiently in Java.

11.9 ***(Elliptic Curve Cryptography)*** Prompt genAIs for code examples demonstrating Java's elliptic curve cryptography capabilities.

11.10 ***(Cryptography Algorithm Comparison)*** Prompt genAIs to

compare the benefits and disadvantages of AES, RSA and Elliptic Curve Cryptography. You can learn about the Java classes and interfaces for this algorithm at

Click here to view code image

```
https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/
   security/interfaces/package-summary.html
```

11.11 ***(File Comparison)*** Prompt genAIs for the algorithm to compare files and determine their differences and how to implement it in Java.

11.12 ***(OpenCSV Library)*** Prompt genAIs for an overview of the OpenCSV library (`https://opencsv.sourceforge.net/`), then ask them to rewrite the code examples in Sections 11.7–11.8 using that library's corresponding capabilities. Also, ask them to provide precise instructions on compiling and running the programs using `javac` and `java` (or your preferred IDE).

11.13 ***(Apache Commons CSV Library)*** Prompt genAIs for an overview of the Apache Commons CSV library (`https://commons.apache.org/proper/commons-csv/`), then ask them to rewrite the code examples in Sections 11.7–11.8 using that library's corresponding capabilities. Also, ask them to provide precise instructions on compiling and running the programs using `javac` and `java` (or your preferred IDE).

Generic Collections

Objectives

In this chapter, you'll:

- Learn what collections are.
- Learn the type-wrapper classes that enable programs to process primitive data values as objects.
- Understand the automatic boxing and unboxing that occurs between objects of the type-wrapper classes and their corresponding primitive types.
- Use prebuilt generic data structures from the collections framework.
- Use various algorithms of the `Collections` class to process collections.
- Use iterators to traverse a collection.
- Learn about factory methods for conveniently creating immutable `List`s, `Set`s and `Map`s optimized for 0 to 10 elements.

Outline

12.1 Introduction

12.2 Collections Overview

12.3 Type-Wrapper Classes

12.4 Boxing and Unboxing

12.5 Lists

12.5.1 `ArrayList` and `Iterator`

12.5.2 `LinkedList`

12.5.3 Views into Collections and Arrays Method `asList`

12.6 `Collections` Methods

12.6.1 Method `sort`

12.6.2 Method `shuffle`

12.6.3 Methods `reverse`, `fill`, `copy`, `max` and `min`

12.6.4 Method `binarySearch`

12.6.5 Methods `addAll`, `frequency` and `disjoint`

12.7 Class `PriorityQueue` and Interface `Queue`

12.8 Hash Tables
12.9 Sets
12.10 Maps
12.11 Convenience Factory Methods for Creating Immutable Collections
12.12 Concurrent Collections
12.13 Wrap-Up
Special Section: Generative AI Exercises

12.1 Introduction

Section 6.20 introduced the generic `ArrayList` collection—a dynamically resizable array-like data structure that stores references to objects of the type you specify when you create the `ArrayList`. Here, we continue our discussion of the **Java collections framework**, which contains many other prebuilt generic data structures. Some examples of collections are the songs stored on a smartphone, a contacts list, the cards a player holds in a card game and the players on a sports team.

We discuss the collections-framework interfaces that declare the capabilities of each collection type, various classes that implement these interfaces, methods that process collection objects, and **iterators** that traverse collections. In Section 12.11, we introduce convenience factory methods that help you create small immutable collections. A factory method is typically a `static` method that builds and returns an object for you.

After reading Chapter 14, Functional Programming with Lambdas & Streams, you'll be able to reimplement many of this chapter's examples more concisely and elegantly, and in a way that makes them easier to parallelize to improve performance on multi-core systems. In Chapter 18, Concurrency: Platform Threads to Virtual Threads, you'll learn how to improve performance on multi-core systems using Java's concurrent collections and parallel stream operations.

Displaying Collections

In this chapter, we sometimes iterate over a collection using a loop to display its elements so we can introduce various collection methods. You can also print a collection's `String` representation. Each Java collection provides a `toString` method that formats the collection's data as a comma-delimited list in square brackets (`[]`) or curly braces (`{}`), depending on the collection's type.

12.2 Collections Overview

A collection (also called a data structure) is an object that can hold references to other objects. Collections contain references to objects of any type that has the *is-a* relationship with the collection's element type.

Java Collections Framework Interfaces

Sections 9.11–9.13 introduced interfaces. The collections framework's interfaces declare the operations performed on various collection types. The following list contains some key collections-framework interfaces[1]—several implementations of these interfaces are provided within the framework, and you may also provide your own:

1. The complete list of collections-framework interfaces can be found at `https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/util/package-summary.html#class-summary` by clicking the Interfaces tab.

   - **`Collection`** is the collections hierarchy's root interface. Interfaces `Set`, `Queue` and `List` extend `Collection`, which contains common collection manipulations, such as adding elements, removing elements, clearing an entire collection, comparing entire collections and converting a collection to an array. In addition, interface `Collection` provides a method that returns an **`Iterator` object** (Section 12.5.1) that a program can use to traverse a collection one element at a time. For the complete list of the `Collection` interface's methods, visit:

     Click here to view code image

     ```
     https://docs.oracle.com/en/java/javase/23/docs/api/java.base/
     ```

- The **`Set` interface** specifies operations for a collection that does not allow duplicate elements. By default, there's no guaranteed order for a `Set`'s elements. However, Java also provides `Set` implementations that maintain elements in their insertion order (the order you add them to the `Set`) or sorted order.
- The **`List` interface** specifies operations for an ordered collection of elements that may contain duplicate elements. A `List` is "ordered" in that you can access its elements by index, like an array.
- The **`Map` interface** specifies operations for a collection that associates keys to values and may not contain duplicate keys. `Map` does not derive from `Collection`. By default, there's no guaranteed order for a `Map`'s key-value pairs. However, Java also provides `Map` implementations that maintain the key-value pairs in their insertion order (the order you add them to the `Map`) or sorted order by key.
- The **`Queue` interface** specifies operations for a first-in, first-out collection that models a waiting line. Other orders can be specified.

Generic Collections

`Collection`, `List`, `Set`, `Map` and `Queue` are generic interfaces. Generics (like `ArrayList` in Section 6.20) enable you to specify the exact type stored in a collection and give you the benefits of compile-time type checking. The compiler issues error messages if you use inappropriate types in your generic collections. We discuss Java's generics mechanisms in more detail in Chapter 13, Generic Classes and Methods: A Deeper Look.

Choosing a Collection

Before building your own data structures, consider using the interfaces and collections of the Java collections framework, which have been carefully tested and tuned to meet most application requirements. The documentation for each collection discusses its memory requirements and the performance characteristics of its methods for operations such as adding and removing elements, searching for elements, sorting elements and more. Chapter 22, Recursion and Big *O*, discusses a means for describing how hard an

algorithm works to perform its task—based on how sensitive it is to the number of data items to be processed. After reading Chapter 22, you'll better understand each collection's performance characteristics described in the online documentation.

Declaring Parameters with Generic Interface Types

Generic interfaces are frequently used as parameter types to allow polymorphic processing of all objects that implement a given generic interface with the correct element type. For example, many Java collection classes have a constructor that takes a `Collection` argument, allowing a new collection to be populated with the elements of any existing collection with the correct element type. So, if you're constructing an `ArrayList<String>`, any object that *is a* `Collection<String>` can be passed to the `ArrayList<String>`'s constructor.

`Collections` Class `static` Methods

Class `Collections` provides `static` convenience methods that search, sort and perform many other common operations on various collections. Section 12.6 discusses several of these `Collections` methods.

Generative AI

1 Prompt genAIs, asking how the Java compiler ensures that only objects of the correct type are added to a generic collection.

12.3 Type-Wrapper Classes

For each primitive type, the `java.lang` package contains a corresponding **type-wrapper class**—**`Boolean`**, **`Byte`**, **`Character`**, **`Double`**, **`Float`**, **`Integer`**, **`Long`** and **`Short`**. These enable programs to manipulate primitive-type values as immutable objects. Java collections cannot store primitive-type values directly, but they can manipulate objects of the type-wrapper classes. The `Byte`, `Short`, `Integer`, `Long`, `Float` and `Double` numeric type-wrapper classes extend **class `Number`**. The type-wrapper classes are `final`, so you cannot extend them.

Unlike class types, primitive types cannot have methods. So, methods associated with a given primitive type are defined as

`static` methods in the corresponding type-wrapper class. For example, class `Integer` contains the `static` method `parseInt`, which converts a `String` to an `int` value. The other type-wrapper classes have similar methods.

Generative AI

1 Prompt genAIs, asking for overviews of commonly used methods provided by each Java type-wrapper class.
2 Prompt genAIs, asking why Java collections cannot store primitive-type values directly.

12.4 Boxing and Unboxing

Java can automatically convert between primitive-type values and type-wrapper objects when necessary:

- A **boxing conversion** converts a primitive-type value to an object of the corresponding type-wrapper class.
- An **unboxing conversion** converts an object of a type-wrapper class to its corresponding primitive-type value.

The following statement boxes the `int` value `10` by creating an `Integer` object containing the value 10 and assigning the object's reference to `integerObject`:

Click here to view code image

```
Integer integerObject = 10; // boxes 10 to an Integer object
```

The following statement unboxes the `int` value `10` from the `Integer` object referenced by `integerObject`:

Click here to view code image

```
int value = integerObject; // unboxes 10 from integerObject
```

Many of our forthcoming examples use boxing and unboxing conversions to store primitive values in and retrieve them from collections.

Generative AI

1 Prompt genAIs, asking about the performance costs of Java's boxing and unboxing.

12.5 Lists

A `List` (sometimes called a **sequence**) is a `Collection` of elements in sequence that may contain duplicate items. Like array indices, `List` indices start at `0`. In addition to the methods inherited from `Collection`, `List` inherits interface `Collection`'s methods and provides additional methods for

- manipulating elements via their indices,
- manipulating a specified range of elements,
- searching for elements and
- obtaining a **`ListIterator`** to access elements.

For the complete list of methods, visit:

Click here to view code image

```
https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/
   util/List.html
```

Several Java Collections Framework classes implement `List`, including `ArrayList` (introduced in Section 6.20) and **`LinkedList`**:

- An `ArrayList` provides a resizable array implementation. Inserting an element between existing `ArrayList` elements is inefficient. All elements after the new one must be moved out of the way, which is expensive for a large `ArrayList`.
- A `LinkedList` enables efficient insertion or removal of elements anywhere in a list but is much less efficient than an `ArrayList` for jumping to an element at a specific index.

12.5.1 **`ArrayList`** and **`Iterator`**

Figure 12.1 uses an `ArrayList` to demonstrate several `Collection` capabilities using two `ArrayList<String>` objects. We broke the example into parts for discussion purposes.

Click here to view code image

```
1  // Fig. 12.1: CollectionTest.java
2  // Collection interface demonstrated via an ArrayList object.
3  import java.util.List;
4  import java.util.ArrayList;
5  import java.util.Collection;
6  import java.util.Iterator;
7
8  public class CollectionTest {
9     public static void main(String[] args) {
```

Fig. 12.1 | `Collection` interface demonstrated via an `ArrayList` object.

Creating the `ArrayList`s and Adding Elements

Lines 11 and 19 declare and initialize `String` arrays `colors` and `removeColors`. Lines 12 and 20 create `ArrayList<String>` objects and assign their references to `List<String>` variables `colorList` and `removeList`—we'll say more about declaring these variables using the `List` interface rather than the `ArrayList` class shortly. Recall that `ArrayList` is a generic class, so we specify a type argument (`String`) to indicate each list's element type. Because you specify the type to store in a collection at compile time, generic collections provide compile-time type safety—the compiler catches attempts to use invalid types. For example, you cannot store `String` objects in a collection of `Employee`s.

Click here to view code image

```
10        // add elements in colors array to colorList
11        String[] colors = {"MAGENTA", "RED", "WHITE", "BLUE", "CYAN"};
12        List<String> colorList = new ArrayList<String>();
13
14        for (String color : colors) {
15           colorList.add(color); // adds color to end of colorList
16        }
17
18        // add elements in removeColors array to removeList
19        String[] removeColors = {"RED", "WHITE", "BLUE"};
20        List<String> removeList = new ArrayList<String>();
21
22        for (String color : removeColors) {
23           removeList.add(color);
24        }
25
```

`List` method `add` appends an element to a `List`. Lines 14–16

use `add` to populate `colorList` with `String`s from array `colors`. Lines 22–24 use `add` to populate `removeList` with `String`s from array `removeColors`. In Section 12.5.3, you'll learn that you can initialize a `List` with all of an array's elements at once.

Displaying the Elements

Lines 29–31 output `colorList`'s elements. Line 29 calls **`List` method `size`** to get the `ArrayList`'s number of elements. Line 30 uses **`List` method `get`** to retrieve elements at each index in the `ArrayList`. We use methods `size` and `get` here for demonstration purposes. Typically, lines 29–31 would be implemented with an enhanced `for` statement, as in lines 39–41. You can also simply print the collection's `String` representation.

Click here to view code image

```
26          // output colorList contents
27          System.out.println("ArrayList: ");
28
29          for (int count = 0; count < colorList.size(); ++count) {
30             System.out.printf("%s ", colorList.get(count));
31          }
32
```

```
ArrayList:
MAGENTA RED WHITE BLUE CYAN
```

Removing Elements

Line 34 calls method `removeColors` (lines 45–56), passing `colorList` and `removeList` as arguments. Method `removeColors` deletes the `String`s in `removeList` from the `String`s in `colorList`. Lines 39–41 print `colorList`'s remaining elements.

Click here to view code image

```
33          // remove from colorList the colors contained in removeList
34          removeColors(colorList, removeList);
35
36          // output colorList contents
37          System.out.printf("%n%nArrayList after calling removeColors:%n");
38
39          for (String color : colorList) {
40             System.out.printf("%s ", color);
41          }
```

```
42      }
43
```

```
ArrayList after calling removeColors:
MAGENTA CYAN
```

Working with Iterators

Method `removeColors` declares two `Collection<String>` parameters (lines 45–46)—any two `Collection`s containing `String`s can be passed as arguments. The method accesses the elements of the first `Collection` (`allItems`) via an `Iterator`. Line 48 calls **`Collection` method `iterator`** to get an `Iterator<String>` for `allItems`. The loop continues executing while **`Iterator` method `hasNext`** (line 51) returns `true`, indicating there are more elements in `allItems` to process.

Click here to view code image

```
44      // remove colors specified in itemsToRemove from allItems
45      private static void removeColors(Collection<String> allItems,
46         Collection<String> itemsToRemove) {
47
48         Iterator<String> iterator = allItems.iterator(); // get iterator
49
50         // loop while collection has items
51         while (iterator.hasNext()) {
52            if (itemsToRemove.contains(iterator.next())) {
53               iterator.remove(); // remove current element
54            }
55         }
56      }
57   }
```

The `if` condition in line 52 calls **`Iterator` method `next`** to obtain a reference to the next element in `allItems`, then uses **`Collection` method `contains`** to determine whether that element is in `itemsToRemove`. If so, line 53 calls **`Iterator` method `remove`** to remove `allItems`' current element.

We declared method `removeColors` in this example to demonstrate working directly with `Iterator`s. However, **`Collections` method `removeAll`** performs the same task. So, the call to `removeColors` in line 34 and its definition in lines 45–56 can be replaced with:

Click here to view code image

```
colorList.removeAll(removeList);
```

In Chapter 14, Functional Programming with Lambdas & Streams, we'll demonstrate filtering, which enables you to produce a new collection containing only the elements that match the criteria you specify—the original collection remains unmodified in the spirit of immutability. In a genAI question in Chapter 14, we'll ask you to prompt genAIs with Fig. 12.1's code, asking them to refactor the code, improving it using techniques introduced in that chapter.

Iterator Invalidation

If a `Collection` is modified by one of its methods after creating an `Iterator` for it, the `Iterator` immediately becomes invalid—any subsequent operation performed with the iterator fails immediately, throwing a `ConcurrentModificationException`. This is known as **failing fast**—fail-fast iterators help ensure that a modifiable collection is not manipulated by multiple threads simultaneously, which could corrupt the data. In Chapter 18, Concurrency: Platform Threads to Virtual Threads, you'll learn about the `java.util.concurrent` package's concurrent collections that multiple concurrent threads can safely manipulate. Modifying the collection via the `Iterator`, however, does not invalidate it.

Referring to Collection Objects with Interface Variables

In this example, we refer to the `ArrayList`s using `List` variables. This makes our code more flexible and easier to modify. If we later determine that `LinkedList`s would be more appropriate, only the `import` statement at line 4 and the statements that created the `ArrayList` objects (lines 12 and 20) need to be modified. In general, when you create a collection object, refer to it with a variable of the corresponding collection interface type. Similarly, implementing method `removeColors` to receive `Collection<String>` parameters enables the method to be used with any object that implements `Collection<String>`.

Type Inference with the `<>` Notation

Lines 12 and 20 specify the `ArrayList`'s element type (`String`) in the variables' types and object creation expressions. You can use type inferencing with the diamond notation `<>` in statements that

declare and create generic type variables and objects. For example, line 12 can be written as:

Click here to view code image

```
List<String> list = new ArrayList<>();
```

The `<>` tells the compiler to infer the `ArrayList`'s element type from the type in angle brackets on the left of the declaration (that is, `String`). We'll use this syntax as appropriate for the remaining examples in this chapter.

Generative AI

1 Prompt genAIs, asking how using variables of Java collection interface types to refer to collection objects makes your code more flexible.

12.5.2 `LinkedList`

Figure 12.2 demonstrates various operations on `LinkedList`s. The program creates two `LinkedList`s of `String`s. We append the elements of one `List` to the other. Then, we convert all the `String`s to uppercase and delete a range of elements. We broke the program into parts for discussion purposes.

Click here to view code image

```
1  // Fig. 12.2: ListTest.java
2  // Lists, LinkedLists and ListIterators.
3  import java.util.List;
4  import java.util.LinkedList;
5  import java.util.ListIterator;
6
7  public class ListTest {
8     public static void main(String[] args) {
```

Fig. 12.2 | `List`s, `LinkedList`s and `ListIterator`s.

Lines 12 and 21 create `LinkedList<String>`s `list1` and `list2`. Like `ArrayList`, `LinkedList` is a generic class with one type parameter (`String` in this example). Lines 14–16 and 23–25 call `List` method `add` to append elements from arrays `colors` and `colors2` to the ends of `list1` and `list2`, respectively.

Click here to view code image

```
9          // add colors elements to list1
10         String[] colors =
11            {"black", "yellow", "green", "blue", "violet", "silver"};
12         List<String> list1 = new LinkedList<>();
13
14         for (String color : colors) {
15            list1.add(color);
16         }
17
18         // add colors2 elements to list2
19         String[] colors2 =
20            {"gold", "white", "brown", "blue", "gray", "silver"};
21         List<String> list2 = new LinkedList<>();
22
23         for (String color : colors2) {
24            list2.add(color);
25         }
26
```

Line 27 calls **`List` method `addAll`** to append all elements of `list2` to the end of `list1`. Line 29 calls method `printList` (lines 41–49) to output `list1`'s contents—again, we can also do this by printing a `List`'s string representation.

Click here to view code image

```
27         list1.addAll(list2); // concatenate lists
28
29         printList(list1);
30
```

```
list:
black yellow green blue violet silver gold white brown blue gray silver
```

Line 31 calls method `convertToUppercaseStrings` (lines 52–59) to convert each `String` element to uppercase, then line 32 calls `printList` again to display the modified `String`s.

Click here to view code image

```
31         convertToUppercaseStrings(list1);
32         printList(list1);
33
```

```
list:
BLACK YELLOW GREEN BLUE VIOLET SILVER GOLD WHITE BROWN BLUE GRAY SILVER
```

Line 35 calls method `removeItems` (lines 62–65) to remove the range of elements starting at index `4` up to, but not including, index `7` of the list. Line 36 displays the updated list, then line 37 calls method `printReversedList` (lines 68–77) to print the list in reverse order.

Click here to view code image

```
34          System.out.printf("%nDeleting elements 4 to 6...");
35          removeItems(list1, 4, 7); // remove items 4-6 from list
36          printList(list1);
37          printReversedList(list1);
38       }
39
```

```
Deleting elements 4 to 6...
list:
BLACK YELLOW GREEN BLUE WHITE BROWN BLUE GRAY SILVER

Reversed list:
SILVER GRAY BLUE BROWN WHITE BLUE GREEN YELLOW BLACK
```

Method `printList`

Method `printList` (lines 41–49) uses an enhanced `for` loop to iterate through its argument's elements and display them.

Click here to view code image

```
40       // output List contents
41       private static void printList(List<String> list) {
42          System.out.printf("%nlist:%n");
43
44          for (String color : list) {
45             System.out.printf("%s ", color);
46          }
47
48          System.out.println();
49       }
50
```

Method `convertToUppercaseStrings`

Method `convertToUppercaseStrings` (lines 52–59) changes lowercase `String` elements in its `List` argument to uppercase. Line 53 calls **`List` method `listIterator`** to get the `List`'s **bidirectional**

iterator, which can traverse a `List` forward or backward and set `List` elements. `ListIterator` is also a generic interface. In this example, the `ListIterator` references `String` objects because method `listIterator` is called on a `List<String>`. Line 55 calls method `hasNext` to determine whether the `List` contains another element. Line 56 gets the next `String` in the `List`. Line 57 calls `String` method `toUpperCase` to get an uppercase `String` and calls **`ListIterator` method `set`** to replace the current `String` to which `iterator` refers with the uppercase `String`.

Click here to view code image

```
51      // locate String objects and convert to uppercase
52      private static void convertToUppercaseStrings(List<String> list) {
53         ListIterator<String> iterator = list.listIterator();
54
55         while (iterator.hasNext()) {
56            String color = iterator.next(); // get item
57            iterator.set(color.toUpperCase()); // convert to upper case
58         }
59      }
60
```

In Chapter 14, Functional Programming with Lambdas & Streams, we'll demonstrate mapping, which enables you to produce a new collection containing a modified copy of each element from the original collection—the original collection remains unmodified in the spirit of immutability. In a genAI question in Chapter 14, we'll ask you to prompt genAIs with Fig. 12.2's code, asking them to refactor the code, improving it using techniques introduced in that chapter.

Method `removeItems`

Method `removeItems` (lines 62–65) removes a range of items from the list. Line 64 calls **`List` method `subList`** to obtain a portion of the `List` (called a **sublist**). This is called a **range-view method**—it enables the program to view a portion of the list. The sublist is simply a view into the `List` on which `subList` is called—the view does not maintain itsown copies of the data. Method `subList` takes two index arguments and returns a sublist from the first index up to, but not including, the second index. In this example, line 64 returns a sublist containing references to the elements at index `4` through `6`—that is, up to, but not including, the element at index `7`. Next, the program calls **`List` method `clear`** (line 64) on the

sublist, which removes the sublist's elements from the `List`. Changes to a sublist modify the original `List` because the sublist does not have copies of the data.

Click here to view code image

```
61      // obtain sublist and use clear method to delete sublist items
62      private static void removeItems(List<String> list,
63         int start, int end) {
64         list.subList(start, end).clear(); // remove items
65      }
66
```

Method `printReversedList`

Method `printReversedList` (lines 68–77) prints the list backward. Line 69 calls `List` method `listIterator` with the starting index position as an argument to get a bidirectional `ListIterator` for the list—such iterators can be initialized with a starting index and provide methods for moving backward through the list. `List` method `size` returns the `List`'s number of items—so the iterator for traversing the list backward is initially positioned one past the last element. The `while` condition (line 74) calls **`ListIterator`'s `hasPrevious` method** to determine whether there are more elements while traversing the list backward. If so, line 75 calls **`ListIterator`'s `previous` method** to get the previous element then displays it.

Click here to view code image

```
67      // print reversed list
68      private static void printReversedList(List<String> list) {
69         ListIterator<String> iterator = list.listIterator(list.size());
70
71         System.out.printf("%nReversed list:%n");
72
73         // print list in reverse order
74         while (iterator.hasPrevious()) {
75            System.out.printf("%s ", iterator.previous());
76         }
77      }
78   }
```

Generative AI

1 Prompt genAIs, asking them to discuss the relative performance of `ArrayList` vs. `LinkedList` without mentioning complexity. Also ask when to use each.

2 Class `LinkedList` also implements the `Deque` interface. Prompt genAIs, asking them what a deque is and what `Deque` methods class `LinkedList` implements. Also, ask for and study a simple Java code example of using a `LinkedList` as a `Deque`.

12.5.3 Views into Collections and `Arrays` Method `asList`

`Arrays` `static` method `asList` returns a `List` view of an array's data, enabling you to use an array in code where a `List` is expected and, since a `List` *is a* `Collection`, also where a `Collection` is expected. For example, we can use a `List` view of an array to

- initialize a new collection with the array's elements by calling the new collection's constructor that receives a `Collection` argument and
- add all of an array's elements to a `Collection` by calling `Collection` method `addAll`, which receives a `Collection` argument.

The next example creates a `LinkedList` and initializes it with a `List` view of an array. Any modifications made to the `List` view change the array's elements, and any modifications made to the array's elements change the `List` view. Recall that an array has a fixed size, so for a `List` view of an array, calling methods that change the `List`'s size by adding or removing elements results in **`UnsupportedOperationException`**s.

Viewing Arrays as `List`s and Converting `List`s to Arrays

Figure 12.3 uses `Arrays` method `asList` to view an array as a `List` and **`List` method `toArray`** to create an array from the contents of a `LinkedList`.

Click here to view code image

```
1  // Fig. 12.3: UsingToArray.java
2  // Viewing arrays as Lists and converting Lists to arrays.
3  import java.util.LinkedList;
```

```
4  import java.util.Arrays;
5
6  public class UsingToArray {
7     public static void main(String[] args) {
8        String[] colors = {"black", "blue", "yellow"};
9        LinkedList<String> list = new LinkedList<>(Arrays.asList(colors));
10
11       list.addLast("red"); // add as last item
12       list.add("pink"); // add to the end
13       list.add(3, "green"); // add at 3rd index
14       list.addFirst("cyan"); // add as first item
15
16       // get LinkedList elements as an array
17       colors = list.toArray(new String[list.size()]);
18
19       System.out.println("colors: ");
20
21       for (String color : colors) {
22          System.out.println(color);
23       }
24    }
25 }
```

```
colors:
cyan
black
blue
yellow
green
red
pink
```

Fig. 12.3 | Viewing arrays as `List`s and converting `List`s to arrays.

Line 9 constructs a `LinkedList<String>` containing the elements of array `colors`. `Arrays` method `asList` returns a `List` view of `colors`, which we pass to the `LinkedList`'s constructor to initialize it. This constructor receives a `Collection` as an argument—again, a `List` *is a* `Collection`. Line 11 calls **`LinkedList` method `addLast`** to append `"red"` to the `list`. Lines 12–13 call **`LinkedList` method `add`** to add `"pink"` as the last element and `"green"` as the element at index `3` (i.e., the fourth element). Method `addLast` works identically to the single-argument `add` method. Line 14 calls **`LinkedList` method `addFirst`** to add `"cyan"` as the new first item in the `LinkedList`. The `add` operations are permitted because they operate on the `LinkedList` object, not the view returned by `asList`. When we add `"cyan"` as the first element, `"green"` becomes the

`LinkedList`'s fifth element.

Line 17 calls the `List` interface's `toArray` method to get a `String` array from `list`. **The array is a copy of the list's elements—modifying the array's contents does not alter the list**. The array passed to `toArray` must have the element type that `toArray` should return. If the array's number of elements exceeds the `LinkedList`'s, `toArray` copies the list's elements into its array argument and returns that array. If the `LinkedList` has more elements than the array argument, `toArray` allocates a new array of the same type, then copies the `list`'s elements into the new array and returns it. So, passing a correctly sized array is more efficient because it avoids reallocation.

Possible `toArray` Logic Error

Passing an array that contains data as `toArray`'s argument can cause logic errors. If the array's number of elements is smaller than the number of elements in the list on which `toArray` is called, a new array is allocated to store the list's elements. If the array's number of elements exceeds the list's, the array's elements (starting at index zero) are overwritten with the list's elements. The first element of the remainder of the array is set to `null` to indicate the end of the list.

Generative AI

1 Prompt genAIs with Fig. 12.1's Java code, asking them to remove the loops that repeatedly call `add` on `colorList` and `removeList` by initializing the lists using `Arrays` method `asList` and the `ArrayList` constructor that receives a `Collection`. Compare the generated code to Fig. 12.1 to ensure the loops were removed.

12.6 `Collections` Methods

Class `Collections` provides many prepackaged, high-performance algorithms for manipulating collection elements. The algorithms in the following table are `static` methods:

`Collections` method	Description
`sort`	Sorts the elements of a `List`.
`binarySearch`	Locates an object in a `List` using the efficient binary search algorithm, which we introduced in Section 6.19.

reverse	Reverses the elements of a `List`.
shuffle	Randomly orders a `List`'s elements.
fill	Sets every `List` element to refer to a specified object.
copy	Copies references from one `List` into another.
min	Returns the smallest element in a `Collection`.
max	Returns the largest element in a `Collection`.
addAll	Appends all elements in an array to a `Collection`.
frequency	Calculates how many `Collection` elements are equal to the specified element.
disjoint	Determines whether two `Collection` have no elements in common.

The methods `sort`, `binarySearch`, `reverse`, `shuffle`, `fill` and `copy` operate on `List`s. Methods `min`, `max`, `addAll`, `frequency` and `disjoint` operate on `Collection`s, so they support a wider range of collection objects than the `List`-specific methods. The `Collections` methods are polymorphic—each can operate on objects that implement specific interfaces, regardless of the underlying implementations.

12.6.1 Method `sort`

Method `sort` sorts the elements of a `List`. The elements' type must implement **interface `Comparable`**. The order is determined by the natural order of the elements' type as implemented by a `compareTo` method. For example, the natural order for numeric values is ascending order, and the natural order for `String`s is based on their lexicographical ordering (Section 7.3). Method `compareTo` is declared in interface `Comparable` and is sometimes called the **natural comparison method**. The `sort` call may specify as a second argument an object that implements **interface `Comparator`** to specify an alternative ordering of the elements.

Sorting in Ascending Order

Figure 12.4 uses `Collections` method `sort` to order the elements of a `List` in ascending order (line 15). Line 12 creates a `List` of `String`s. Lines 13 and 16 each use an implicit call to the `list`'s `toString` method to output the list contents as a comma-delimited list in square brackets (`[` and `]`).

Click here to view code image

```
1  // Fig. 12.4: Sort1.java
2  // Collections method sort.
3  import java.util.List;
4  import java.util.Arrays;
5  import java.util.Collections;
6
7  public class Sort1 {
8     public static void main(String[] args) {
9        String[] suits = {"Hearts", "Diamonds", "Clubs", "Spades"};
10
11       // Create and display a list containing the suits array elements
12       List<String> list = Arrays.asList(suits);
13       System.out.printf("Unsorted array elements: %s%n", list);
14
15       Collections.sort(list); // sort ArrayList
16       System.out.printf("Sorted array elements: %s%n", list);
17    }
18 }
```

```
Unsorted array elements: [Hearts, Diamonds, Clubs, Spades]
Sorted array elements: [Clubs, Diamonds, Hearts, Spades]
```

Fig. 12.4 | `Collections` method `sort`.

Sorting in Descending Order

Figure 12.5 sorts the same list of strings used in Fig. 12.4 in descending order. The example introduces the `Comparator` interface to customize sorting a `Collection`'s elements. Line 16 calls `Collections`'s method `sort` to order the `List` in descending order. The **`static Collections` method `reverseOrder`** returns a `Comparator` object that orders the collection's elements in reverse order. Because the collection being sorted is a `List<String>`, `reverseOrder` returns a `Comparator<String>`.

Click here to view code image

```
1  // Fig. 12.5: Sort2.java
2  // Using a Comparator object with method sort.
3  import java.util.List;
4  import java.util.Arrays;
5  import java.util.Collections;
6
7  public class Sort2 {
8     public static void main(String[] args) {
9        String[] suits = {"Hearts", "Diamonds", "Clubs", "Spades"};
10
```

```
        // Create and display a list containing the suits array elements
        List<String> list = Arrays.asList(suits); // create List
        System.out.printf("Unsorted array elements: %s%n", list);

        // sort in descending order using a comparator
        Collections.sort(list, Collections.reverseOrder());
        System.out.printf("Sorted list elements: %s%n", list);
    }
}
```

```
Unsorted array elements: [Hearts, Diamonds, Clubs, Spades]
Sorted list elements: [Spades, Hearts, Diamonds, Clubs]
```

Fig. 12.5 | `Collections` method `sort` with a `Comparator` object.

Sorting with a Custom `Comparator`

Figure 12.6 creates a custom class `TimeComparator` that implements interface `Comparator` to compare two `Time2` objects. Recall that class `Time2` (Fig. 8.14) represents times with hours, minutes and seconds.

Click here to view code image

```
// Fig. 12.6: TimeComparator.java
// Custom Comparator class that compares two Time2 objects.
import java.util.Comparator;

public class TimeComparator implements Comparator<Time2> {
   @Override
   public int compare(Time2 time1, Time2 time2) {
      int hourDifference = time1.getHour() - time2.getHour();

      if (hourDifference != 0) { // test the hour first
         return hourDifference;
      }

      int minuteDifference = time1.getMinute() - time2.getMinute();

      if (minuteDifference != 0) { // then test the minute
         return minuteDifference;
      }

      int secondDifference = time1.getSecond() - time2.getSecond();
      return secondDifference;
   }
}
```

Fig. 12.6 | Custom `Comparator` class that compares two `Time2` objects.

Class `TimeComparator` (Fig. 12.6) implements generic interface `Comparator`, which has one type argument (in this case, `Time2`). A class that implements `Comparator` must declare a `compare` method that receives two arguments of the type to compare and returns

- a negative integer if the first argument is less than the second,
- 0 if the arguments are equal or
- a positive integer if the first argument is greater than the second.

Method `compare` (lines 6–22) compares `Time2` objects. Line 8 calculates the difference between their hours. If the hours are different (line 10), then we return this value:

- If this value is positive, `time1`'s hour is greater than `time2`'s, and `time1` is greater than `time2`.
- If this value is negative, `time1`'s hour is less than `time2`'s, and `time1` is less than `time2`.

If the hour difference is zero, the hours are the same, so we must test the minutes (lines 14–18) to determine which time is greater. If the minutes are the same, we must test the seconds (lines 20–21) to determine which time is greater.

Figure 12.7 sorts an `ArrayList<Time2>` object's elements using our `TimeComparator`. Line 9 creates the `ArrayList`. Lines 11–15 create five `Time2` objects and add them to the list. Line 21 calls method `sort`, passing it a `TimeComparator` object to customize the comparison (Fig. 12.6).

Click here to view code image

```
1  // Fig. 12.7: Sort3.java
2  // Collections method sort with a custom Comparator object.
3  import java.util.List;
4  import java.util.ArrayList;
5  import java.util.Collections;
6
7  public class Sort3 {
8     public static void main(String[] args) {
9        List<Time2> list = new ArrayList<>(); // create List
10
```

```
        list.add(new Time2(6, 24, 34));
        list.add(new Time2(18, 14, 58));
        list.add(new Time2(6, 5, 34));
        list.add(new Time2(12, 14, 58));
        list.add(new Time2(6, 24, 22));

        // output List elements
        System.out.printf("Unsorted array elements:%n%s%n", list);

        // sort in order using a comparator
        Collections.sort(list, new TimeComparator());

        // output List elements
        System.out.printf("Sorted list elements:%n%s%n", list);
    }
}
```

```
Unsorted array elements:
[6:24:34 AM, 6:14:58 PM, 6:05:34 AM, 12:14:58 PM, 6:24:22 AM]
Sorted list elements:
[6:05:34 AM, 6:24:22 AM, 6:24:34 AM, 12:14:58 PM, 6:14:58 PM]
```

Fig. 12.7 | `Collections` method `sort` with a custom `Comparator` object.

Generative AI

1 Prompt genAIs with Fig. 8.5's `Account` class, asking them to create a `Comparator` that can help sort `Account` objects by their balances. Ask the genAIs to write a test program that creates an `ArrayList<Account>`, populates it with five `Account` objects, then sorts the accounts by their balances in ascending and descending order and displays the results.

12.6.2 Method `shuffle`

Method `shuffle` randomly orders a `List`'s elements. Chapter 8 presented a card shuffling and dealing simulation that shuffled a deck of cards with a loop. Figure 12.8 uses method `shuffle` to shuffle a deck of `Card` objects that might be used in a card-game simulator.

Class `Card` (lines 8–32) represents a card in a deck of cards. Each has a face and a suit. Lines 9–11 declare two `enum` types—`Face` and `Suit`—representing a `Card`'s face and suit. Method `toString` (lines

29–31) returns a `String` containing the `Card`'s face and suit separated by `"of"`. When an `enum` constant is converted to a `String`, its identifier is used as the `String` representation. This example capitalizes only the first letter of each `enum` constant, so a `Card`'s face and suit will be displayed with initial capital letters (e.g., `"Ace of Spades"`).

Click here to view code image

```
1  // Fig. 12.8: DeckOfCards.java
2  // Card shuffling and dealing with Collections method shuffle.
3  import java.util.ArraysList;
4  import java.util.List;
5  import java.util.Collections;
6
7  // class to represent a Card in a deck of cards
8  class Card {
9     public enum Face {Ace, Deuce, Three, Four, Five, Six,
10       Seven, Eight, Nine, Ten, Jack, Queen, King }
11    public enum Suit {Clubs, Diamonds, Hearts, Spades}
12
13    private final Face face;
14    private final Suit suit;
15
16    // constructor
17    public Card(Face face, Suit suit) {
18       this.face = face;
19       this.suit = suit;
20    }
21
22    // return face of the card
23    public Face getFace() {return face;}
24
25    // return suit of Card
26    public Suit getSuit() {return suit;}
27
28    // return String representation of Card
29    public String toString() {
30       return String.format("%s of %s", face, suit);
31    }
32 }
33
34 // class DeckOfCards declaration
35 public class DeckOfCards {
36    private List<Card> deck = new ArrayList<>(); // List to store Cards
37
38    // set up deck of Cards and shuffle
39    public DeckOfCards() {
40       // populate deck with Card objects
```

```
41          for (Card.Suit suit : Card.Suit.values()) {
42             for (Card.Face face : Card.Face.values()) {
43                deck.add(new Card(face, suit));
44             }
45          }
46
47          Collections.shuffle(deck); // shuffle deck
48       }
49
50       // output deck
51       public void printCards() {
52          // display 52 cards in four columns
53          for (int i = 0; i < deck.size(); ++i) {
54             System.out.printf("%-19s%s", deck.get(i),
55                ((i + 1) % 4 == 0) ? System.lineSeparator() :"");
56          }
57       }
58
59       public static void main(String[] args) {
60          DeckOfCards cards = new DeckOfCards();
61          cards.printCards();
62       }
63    }
```

```
Deuce of Clubs     Six of Spades      Nine of Diamonds   Ten of Hearts
Three of Diamonds  Five of Clubs      Deuce of Diamonds  Seven of Clubs
Three of Spades    Six of Diamonds    King of Clubs      Jack of Hearts
Ten of Spades      King of Diamonds   Eight of Spades    Six of Hearts
Nine of Clubs      Ten of Diamonds    Eight of Diamonds  Eight of Hearts
Ten of Clubs       Five of Hearts     Ace of Clubs       Deuce of Hearts
Queen of Diamonds  Ace of Diamonds    Four of Clubs      Nine of Hearts
Ace of Spades      Deuce of Spades    Ace of Hearts      Jack of Diamonds
Seven of Diamonds  Three of Hearts    Four of Spades     Four of Diamonds
Seven of Spades    King of Hearts     Seven of Hearts    Five of Diamonds
Eight of Clubs     Three of Clubs     Queen of Clubs     Queen of Spades
Six of Clubs       Nine of Spades     Four of Hearts     Jack of Clubs
Five of Spades     King of Spades     Jack of Spades     Queen of Hearts
```

Fig. 12.8 | Card shuffling and dealing with `Collections` method `shuffle`.

Lines 41–45 populate the `deck` array with `Card`s having unique `Face` and `Suit` combinations. Both `Face` and `Suit` are nested types in class `Card`. To use these `enum` types outside of class `Card`, you must qualify each `enum`'s type name with the name of the class in which it resides (i.e., `Card`) and a dot (`.`) separator. Hence, lines 41 and 42 use `Card.Suit` and `Card.Face` to declare the control variables of the `for` statements. Recall that an `enum` type's `values`

method returns an array containing the type's constants. Lines 41–45 use enhanced `for` statements to construct 52 new `Card`s. Line 47 calls `static Collections` method `shuffle` to shuffle the `Card`s. Method `shuffle` requires a `List` argument.

Method `printCards` (lines 51–57) displays the deck in four columns. Each loop iteration outputs a card left-aligned in a 19-character field followed by either a newline or an empty string (lines 54–55) based on the number of cards output so far. If that number is divisible by 4, a newline is output. Line 55 uses `System` method `lineSeparator` to get the platform-independent newline character to output after every four cards.

Generative AI

1 Sometimes, you may want a **reproducible** random sequence, as is common in data science applications. For example, each time you execute a card-playing application while you debug its logic, you might want a shuffled deck to have the cards in the same order. Prompt genAIs, asking how to use an overloaded version of `Collections` method `shuffle` and a `RandomGenerator` object to obtain reproducible results.

12.6.3 Methods `reverse`, `fill`, `copy`, `max` and `min`

Class `Collections` provides methods for reversing, filling and copying `List`s:

- **`Collections` method `reverse`** reverses the order of the elements in a `List`.
- **`Collections` method `fill`** overwrites elements in a `List` with a specified value—for example, to reinitialize an existing `List`.
- **`Collections` method `copy`** receives a destination `List` and a source `List`. Each element in the source is copied to the destination. The destination `List` must be at least as long as the source; otherwise, an `IndexOutOfBoundsException` occurs. If the destination `List` is longer, the elements not overwritten are unchanged.

`Collections` methods `min` and `max` operate on any `Collection`—

`min` returns the smallest element, and `max` returns the largest. Both can be called with a `Comparator` object as a second argument to perform custom comparisons of objects, such as the `TimeComparator` in Fig. 12.7. Figure 12.9 demonstrates methods `reverse`, `fill`, `copy`, `max` and `min`.

Click here to view code image

```
1  // Fig. 12.9: Algorithms1.java
2  // Collections methods reverse, fill, copy, max and min.
3  import java.util.List;
4  import java.util.Arrays;
5  import java.util.Collections;
6
7  public class Algorithms1 {
8     public static void main(String[] args) {
9        // create and display a List<Character>
10       Character[] letters = {'P', 'C', 'M'};
11       List<Character> list = Arrays.asList(letters); // get List
12       outputList(list);
13
14       // reverse and display the List<Character>
15       Collections.reverse(list); // reverse order the elements
16       System.out.printf("%nAfter calling reverse, list contains:%n");
17       outputList(list);
18
19       // create copyList from an array of 3 Characters
20       Character[] lettersCopy = new Character[3];
21       List<Character> copyList = Arrays.asList(lettersCopy);
22
23       // copy the contents of list into copyList
24       Collections.copy(copyList, list);
25       System.out.printf("%nAfter copying, copyList contains:%n");
26       outputList(copyList);
27
28       // fill list with Rs
29       Collections.fill(list, 'R');
30       System.out.printf("%nAfter calling fill, list contains:%n");
31       outputList(list);
32    }
33
34    // output List information
35    private static void outputList(List<Character> list) {
36       System.out.printf("The list is: %s%n", list);
37       System.out.printf("Max: %s; Min: %s%n", Collections.max(list),
38          Collections.min(list));
39    }
40 }
```

```
The list is: [P, C, M]
Max: P; Min: C

After calling reverse, list contains:
The list is: [M, C, P]
Max: P; Min: C

After copying, copyList contains:
The list is: [M, C, P]
Max: P; Min: C

After calling fill, list contains:
The list is: [R, R, R]
Max: R; Min: R
```

Fig. 12.9 | `Collections` methods `reverse`, `fill`, `copy`, `max` and `min`.

Line 11 creates the `List<Character>` variable `list` and initializes it with a `List` view of the `Character` array `letters`. Line 12 calls method `outputList` (lines 35–39) to display the current contents of the `List` and its minimum and maximum values. Line 15 calls `Collections` method `reverse` to reverse the order of `list`. Method `reverse` takes one `List` argument. Since `list` is a `List` view of the array `letters`, the array's elements are now in reverse order. The reversed contents are output in lines 16–17.

Line 24 uses `Collections` method `copy` to copy `list`'s elements into `copyList`. Changes to `copyList` do not change `letters` because `copyList` is a separate `List` that's not a `List` view of the array `letters`. Method `copy` requires two `List` arguments—the destination `List` and the source `List`.

Line 29 calls `Collections` method `fill` to place the character `'R'` in each `list` element. Because `list` is a `List` view of the array `letters`, this operation changes each element in `letters` to `'R'`. Method `fill` requires a `List` for the first argument and an object of the `List`'s element type for the second argument. In this case, the object is the boxed `Character` version of `'R'`.

Lines 37–38 in method `outputList` call `Collections` methods `max` and `min` to find the largest and the smallest element of a `Collection`, respectively. Recall that interface `List` extends interface `Collection`, so a `List` is a `Collection`.

Generative AI

1 Prompt genAIs, asking what happens if you call the Java `Collections` class's `min` or `max` methods on a collection whose elements do not implement the `Comparable` interface.

12.6.4 Method `binarySearch`

If you need to search for a key in a short `List`, simply iterating through the `List`'s elements is usually efficient. If you need to search for a key in a lengthy `List`, the high-speed binary search algorithm can be far more efficient. The **`static Collections` method `binarySearch`** implements the high-speed binary search algorithm. This method locates an object in a **`List` that's sorted in ascending order**. If the object is found, `binarySearch` returns its index; otherwise, `binarySearch` returns a negative value. If multiple elements in the list match the search key, there's no guarantee which one will be located first. Figure 12.10 uses `binarySearch` to search for various `String`s in an `ArrayList`.

Click here to view code image

```
1  // Fig. 12.10: BinarySearchTest.java
2  // Collections method binarySearch.
3  import java.util.List;
4  import java.util.Arrays;
5  import java.util.Collections;
6  import java.util.ArrayList;
7
8  public class BinarySearchTest {
9     public static void main(String[] args) {
10       // create an ArrayList<String> from the contents of colors array
11       String[] colors = {"red", "white", "blue", "black","yellow",
12          "purple", "tan", "pink"};
13       List<String> list = new ArrayList<>(Arrays.asList(colors));
14
15       Collections.sort(list); // sort the ArrayList
16       System.out.printf("Sorted ArrayList: %s%n", list);
17
18       // search list for various values
19       printSearchResults(list, "black");
20       printSearchResults(list, "red");
21       printSearchResults(list, "pink");
22       printSearchResults(list, "aqua"); // below lowest
23       printSearchResults(list, "gray"); // does not exist
24       printSearchResults(list, "teal"); // does not exist
25    }
```

```
26
27      // perform search and display result
28      private static void printSearchResults(
29         List<String> list, String key) {
30
31         System.out.printf("%nSearching for: %s%n", key);
32         int result = Collections.binarySearch(list, key);
33
34         if (result >= 0) {
35            System.out.printf("Found at index %d%n", result);
36         }
37         else {
38            System.out.printf("Not Found (%d)%n",result);
39         }
40      }
41   }
```

```
Sorted ArrayList: [black, blue, pink, purple, red, tan, white, yellow]

Searching for: black
Found at index 0

Searching for: red
Found at index 4

Searching for: pink
Found at index 2

Searching for: aqua
Not Found (-1)

Searching for: gray
Not Found (-3)

Searching for: teal
Not Found (-7)
```

Fig. 12.10 | `Collections` method `binarySearch`.

Line 13 initializes `list` with an `ArrayList` containing a copy of the elements in array `colors`. A precondition (Section 10.10) of `Collections` method `binarySearch` is that its `List` argument's elements must be sorted in ascending order. Line 15 uses `Collections` method `sort` to sort the `List`, which is a time-consuming operation for large `List`s. If `binarySearch`'s precondition is not met, its result is undefined—you'll explore this issue with genAI momentarily. Line 16 outputs the sorted list. Lines 19–24 call method `printSearchResults` (lines 28–40) to perform

searches and output the results. Line 32 calls `Collections` method `binarySearch` to search `list` for the specified `key`, passing a `List` and the search key as arguments. Lines 34–39 output the search results. An overloaded version of `binarySearch` takes a `Comparator` object as its third argument, which specifies how `binarySearch` should compare the search key to the `List`'s elements.

Generative AI

1 Business-critical and mission-critical systems should not contain code that could produce undefined results, potentially causing serious problems. Prompt genAIs, asking how to avoid undefined results from the Java `Collections` method `binarySearch`.
2 Prompt genAIs, asking how Java `Collections` method `binarySearch` calculates its negative return value when it does not find the key.

12.6.5 Methods `addAll`, `frequency` and `disjoint`

Class `Collections` also provides the methods `addAll`, `frequency` and `disjoint`:

- **`Collections` method `addAll`** takes two arguments—a `Collection` into which to insert the new element(s) and an array (or variable-length argument list) that provides elements to be inserted.
- **`Collections` method `frequency`** takes two arguments—a `Collection` to search and an `Object` to be searched for in the collection. Method `frequency` returns the number of times the second argument appears in the collection.
- **`Collections` method `disjoint`** takes two `Collection`s and returns `true` if they have no elements in common.

Figure 12.11 demonstrates these methods.

Click here to view code image

```
1  // Fig. 12.11: Algorithms2.java
2  // Collections methods addAll, frequency and disjoint.
3  import java.util.ArrayList;
4  import java.util.List;
```

```
5  import java.util.Arrays;
6  import java.util.Collections;
7
8  public class Algorithms2 {
9     public static void main(String[] args) {
10       // initialize list1 and list2
11       String[] colors = {"red", "white", "yellow", "blue"};
12       List<String> list1 = Arrays.asList(colors);
13       List<String> list2 = new ArrayList<>();
14
15       list2.add("black"); // add "black" to the end of list2
16       list2.add("red"); // add "red" to the end of list2
17       list2.add("green"); // add "green" to the end of list2
18
19       System.out.printf("Before addAll, list2 contains: %s%n", list2);
20       Collections.addAll(list2, colors); // add colors Strings to list2
21       System.out.printf("After addAll, list2 contains: %s%n", list2);
22
23       // get frequency of "red"
24       int frequency = Collections.frequency(list2, "red");
25       System.out.printf("%nFrequency of red in list2: %d%n", frequency);
26
27       // check whether list1 and list2 have any elements in common
28       boolean areDisjoint = Collections.disjoint(list1, list2);
29
30       System.out.printf("list1 and list2 %s elements in common%n",
31          (areDisjoint ? "do not have" : "have"));
32    }
33 }
```

```
Before addAll, list2 contains: [black, red, green]
After addAll, list2 contains: [black, red, green, red, white, yellow, blue]

Frequency of red in list2: 2
list1 and list2 have elements in common
```

Fig. 12.11 | `Collections` methods `addAll`, `frequency` and `disjoint`.

Line 12 initializes `list1` with elements in array `colors`, and lines 15–17 add `String`s `"black"`, `"red"` and `"green"` to `list2`. Line 20 invokes method `addAll` to add elements in array `colors` to `list2`. Line 24 gets the frequency of `"red"` in `list2` using method `frequency`. Line 28 invokes method `disjoint` to test whether `Collection`s `list1` and `list2` have elements in common, which they do in this example.

In Chapter 14, Functional Programming with Lambdas & Streams, a genAI question will ask you to prompt genAIs with Fig.

12.11's code, asking them to refactor the code, improving it using techniques introduced in that chapter.

Generative AI

1 Prompt genAIs, asking them to discuss the performance of `Collections` methods `addAll`, `frequency` and `disjoint` for large collections.

12.7 Class `PriorityQueue` and Interface `Queue`

Recall that a queue is a collection representing a waiting line—typically, insertions are made at the back of a queue, and deletions are made from the front. This section demonstrates Java's `Queue` interface and **`PriorityQueue` class** from package `java.util`. In Chapter 18, Concurrency: Platform Threads to Virtual Threads, we'll use concurrent queues.

Interface `Queue` extends interface `Collection` and provides additional operations for inserting, removing and inspecting elements in a queue. Class `PriorityQueue`, which implements the `Queue` interface, orders elements by their natural ordering as specified by `Comparable` elements' `compareTo` method or by a `Comparator` object supplied to the constructor.

Class `PriorityQueue` provides functionality that enables insertions in sorted order into the underlying data structure and deletions from the front of the underlying data structure. For `PriorityQueue`s containing numeric values, the lowest value has the "highest" priority and will be removed from the `PriorityQueue` first.

`PriorityQueue` Operations

The common `PriorityQueue` operations are:

- **`offer`** to insert an element at the appropriate location based on priority order,
- **`poll`** to remove the highest-priority element of the priority

queue (i.e., the head of the queue),

- **`peek`** to get a reference to the highest-priority element of the priority queue (without removing that element),
- **`clear`** to remove all elements in the priority queue and
- **`size`** to get the number of elements in the priority queue.

Demonstrating `PriorityQueue`

Figure 12.12 demonstrates generic class `PriorityQueue` with a `PriorityQueue<Double>` (line 8). By default a `PriorityQueue`'s initial capacity is `11` elements. Class `PriorityQueue` provides several additional constructors. One of these takes an `int` and a `Comparator` object to create a `PriorityQueue` with the initial capacity specified by the `int` and the ordering by the `Comparator`.

Click here to view code image

```
1  // Fig. 12.12: PriorityQueueTest.java
2  // PriorityQueue test program.
3  import java.util.PriorityQueue;
4
5  public class PriorityQueueTest {
6     public static void main(String[] args) {
7        // queue of capacity 11
8        var queue = new PriorityQueue<Double>();
9
10       // insert elements to queue
11       queue.offer(3.2);
12       queue.offer(9.8);
13       queue.offer(5.4);
14
15       System.out.print("Polling from queue: ");
16
17       // display elements in queue
18       while (!queue.isEmpty()) {
19          System.out.printf("%.1f ", queue.peek()); // view top element
20          queue.poll(); // remove top element
21       }
22    }
23 }
```

```
Polling from queue: 3.2 5.4 9.8
```

Fig. 12.12 | `PriorityQueue` test program.

Lines 11–13 use method `offer` to add elements to the priority

queue. Method `offer` throws a `NullPointerException` if the program attempts to add a `null` object to the queue. The loop in lines 18–21 uses **method `isEmpty`** to determine whether the priority queue is empty (line 18). While there are more elements, line 19 uses `PriorityQueue` method `peek` to look at the highest-priority element in the queue for output (without actually removing it from the queue). Line 20 removes the highest-priority element in the queue with method `poll`, which returns the removed element.

Generative AI

1 Prompt genAIs for examples of priority queues in computer systems.
2 Prompt genAIs, asking when to use a `Queue`'s `offer` and `poll` methods vs. a `Queue`'s `add` and `remove` methods.

12.8 Hash Tables

When a program creates objects, it may need to store and retrieve them efficiently. Storing and retrieving information with arrays is efficient if some aspect of your data matches a numerical key value (that is, the index) and if the keys are unique and tightly packed. If you have 100 employees with nine-digit Social Security numbers and you want to store and retrieve employee data by using the Social Security number as an array index, the task will require an array with approximately 900 million elements because nine-digit Social Security numbers must begin with 001–899 (excluding 666) as per the Social Security Administration's website:

Click here to view code image

```
https://www.ssa.gov/employer/randomization.html
```

A program with such a large array could achieve high performance for storing and retrieving employee records by simply using the Social Security number as the array index. This is impractical for most applications that use Social Security numbers as keys.

Non-Integer and Sparsely-Spread Keys

Numerous applications have this problem—namely, that either the keys are of the wrong type (e.g., not positive integers usable as array subscripts) or they're of the right type but sparsely spread over a vast range, such as Social Security numbers for a small company's employees. What is needed is a high-speed scheme for converting keys—such as Social Security numbers, inventory part numbers and the like—into unique array indices over a modest-size array. Then, when an application needs to store a data item, the scheme can rapidly convert the application's key into an array index (a process called **hashing**), and the item can be stored at that slot in the array. Retrieval is accomplished the same way: Once the application has a key for which it wants to retrieve the data, it simply applies the conversion to the key, producing the array index where the data is stored and retrieved.

Why the Term Hashing?

When we convert a key into an array index, we literally scramble the bits, forming a kind of "mishmashed," or hashed, number. The number actually has no real significance beyond its usefulness in storing and retrieving particular data.

Hashing Collisions

A glitch in the scheme is called a **collision**, which occurs when two different keys "hash into" the same array element (or cell). We cannot store two values in the same space, so we need to find an alternative home for all values beyond the first that hash to the same array index. There are many schemes for doing this. One is to "hash again"—that is, to apply another hashing transformation to the previous hash result to provide the next candidate cell in the array. The hashing process is designed to distribute the values throughout the table, so hopefully, an available cell will be found with one or a few hashes.

Another scheme uses one hash to locate the first candidate cell. If that cell is occupied, adjacent cells are searched sequentially until an available cell is found. Retrieval works the same way: The key is hashed once to determine the initial location and check whether it contains the desired data. If it does, the search is finished; otherwise, successive cells are searched sequentially until the desired data is found. A popular solution to hash-table collisions is to have each table cell be a **hash bucket**—typically a list of all the key–value pairs that hash to that cell. This is the solution that Java's `HashMap` class (Section 12.10) uses.

Load Factor and Performance

A **hash table's load factor** affects the performance of hashing schemes. The load factor is the ratio of the hash table's number of occupied cells to its total number of cells. The closer this ratio gets to 1.0, the greater the chance of collisions, which slow data insertion and retrieval.

A hash table's load factor is a classic example of a memory-space/execution-time tradeoff: By increasing the load factor, we get better memory utilization, but the program runs slower due to increased hashing collisions. By decreasing the load factor, we get better program speed because of reduced hashing collisions, but we get poorer memory utilization because a larger portion of the hash table remains empty.

`HashSet` and `HashMap` Collections

Java's `HashSet` and `HashMap` collections (presented in Sections 12.9–12.10) are implemented as hash tables "under the hood." When you use those collections, you get the benefit of high-speed data storage and retrieval without having to build your own hash-table mechanisms—a classic example of software reuse. Class `Object`'s `hashCode` method is used to determine a key's location in a Java hash-table-based collection.

Virtual Memory

Another interesting application of hashing is with virtual memory[2] operating systems. Programs and data are maintained on massive secondary storage and brought into more limited primary memory (and even more limited and faster cache memory) to execute. The trick to running applications efficiently is to keep in primary memory (and cache memory) the portion of the program that needs to execute at a given time. Finding those program pieces in virtual memory involves searching the large data structures that define it. Hashing is often used for that purpose.

2. "Virtual Memory." Wikipedia. Wikimedia Foundation. Accessed September 24, 2024. https://en.wikipedia.org/wiki/Virtual_memory.

Generative AI

1 Prompt genAIs to briefly overview how hashing is used to

implement high-performance virtual memory operating systems.

12.9 Sets

A `Set` is a collection of unique elements (i.e., no duplicates). The collections framework contains several `Set` implementations, including `HashSet` and `TreeSet`:

- A **`HashSet`** stores its elements (unordered) in a hash table—hash tables are presented in Section 12.8.
- A **`TreeSet`** stores its elements (ordered) in a tree. Trees and their applications are discussed in upper-level computer science courses typically called Data Structures and Algorithms.

Figure 12.13 uses a `HashSet` to remove duplicate strings from a `List`. Recall that both `List` and `Collection` are generic types, so line 14 creates a `List` that contains `String` objects, and line 18 passes the collection to method `printNonDuplicates` (lines 22–33), which takes a `Collection` argument. Line 24 constructs a `HashSet<String>` from the `Collection<String>` argument. By definition, `Set`s do not contain duplicates, so when the `HashSet` is constructed, any duplicates in the `Collection` are ignored. Lines 28–30 output elements in the `Set`. We used a loop here to iterate through a `HashSet`'s elements, but we also could have used the `HashSet`'s `String` representation.

Click here to view code image

```
1  // Fig. 12.13: SetTest.java
2  // HashSet used to remove duplicate values from an array of strings.
3  import java.util.List;
4  import java.util.Arrays;
5  import java.util.HashSet;
6  import java.util.Set;
7  import java.util.Collection;
8
9  public class SetTest {
10    public static void main(String[] args) {
11       // create and display a List<String>
12       String[] colors = {"red", "white", "blue", "green", "gray",
13          "orange", "tan", "white", "cyan", "peach", "gray", "orange"};
```

```
14          List<String> list = Arrays.asList(colors);
15          System.out.printf("List: %s%n", list);
16
17          // eliminate duplicates then print the unique values
18          printNonDuplicates(list);
19       }
20
21       // create a Set from a Collection to eliminate duplicates
22       private static void printNonDuplicates(Collection<String> values) {
23          // create a HashSet
24          Set<String> set = new HashSet<>(values);
25
26          System.out.printf("%nNonduplicates are: ");
27
28          for (String value : set) {
29             System.out.printf("%s ", value);
30          }
31
32          System.out.println();
33       }
34    }
```

```
List: [red, white, blue, green, gray, orange, tan, white, cyan, peach, gray,
orange]

Nonduplicates are: red orange tan green gray white blue peach cyan
```

Fig. 12.13 | `HashSet` used to remove duplicate values from an array of strings.

Sorted Sets

The collections framework also includes the **`SortedSet` interface** (which extends `Set`) for sets that maintain their elements in sorted order—either the elements' natural order (e.g., numbers are in ascending order) or an order specified by a `Comparator`. Class `TreeSet` implements `SortedSet`. Figure 12.14 places `String`s into a `TreeSet`. The `String`s are sorted as they're added to the `TreeSet`. This example also demonstrates range-view methods, which enable a program to view a portion of a collection.

Click here to view code image

```
1  // Fig. 12.14: SortedSetTest.java
2  // Using SortedSets and TreeSets.
3  import java.util.Arrays;
4  import java.util.SortedSet;
```

```
5  import java.util.TreeSet;
6
7  public class SortedSetTest {
8     public static void main(String[] args) {
9        // create TreeSet from array colors
10       String[] colors = {"yellow", "green", "black", "tan", "grey",
11          "white", "orange", "red", "green"};
12       SortedSet<String> tree = new TreeSet<>(Arrays.asList(colors));
13
14       System.out.print("sorted set: ");
15       printSet(tree);
16
17       // get headSet based on "orange"
18       System.out.print("headSet (\"orange\"): ");
19       printSet(tree.headSet("orange"));
20
21       // get tailSet based on "orange"
22       System.out.print("tailSet (\"orange\"): ");
23       printSet(tree.tailSet("orange"));
24
25       // get first and last elements
26       System.out.printf("first: %s%n", tree.first());
27       System.out.printf("last : %s%n", tree.last());
28    }
29
30    // output SortedSet using enhanced for statement
31    private static void printSet(SortedSet<String> set) {
32       for (String s : set) {
33          System.out.printf("%s ", s);
34       }
35
36       System.out.println();
37    }
38 }
```

```
sorted set: black green grey orange red tan white yellow
headSet ("orange"): black green grey
tailSet ("orange"): orange red tan white yellow
first: black
last : yellow
```

Fig. 12.14 | Using `SortedSet`s and `TreeSet`s.

Line 12 creates a `TreeSet<String>` containing the elements of array `colors`, then assigns the new `TreeSet<String>` to `SortedSet<String>` variable `tree`. Line 15 outputs the initial set using method `printSet` (lines 31–37), which we discuss momentarily. Line 19 calls **`TreeSet` method `headSet`** to get a subset of the `TreeSet` in which every element is less than `"orange"`.

The view returned from `headSet` is then output with `printSet`.

Line 23 calls **`TreeSet` method `tailSet`** to get a subset in which each element is greater than or equal to `"orange"`, then outputs the result. Lines 26–27 call **`SortedSet` methods `first`** and **`last`** to get the smallest and largest elements of the set, respectively. Methods `headSet` and `tailSet` return views of the `TreeSet`'s data. Any changes made to the view modify the original `TreeSet`.

Method `printSet` (lines 31–37) accepts a `SortedSet` as an argument and prints it. Lines 32–34 print each element of the `SortedSet`. We used a loop here to iterate through a `SortedSet`'s elements, but we also could have displayed the `SortedSet`'s `String` representation. Note that the loop iterates through the `SortedSet`'s elements in sorted order.

Generative AI

1 Prompt genAIs, asking about the performance of `HashSet` vs. `TreeSet` and when to use each.

12.10 Maps

`Map`s associate keys to values. The keys in a `Map` must be unique, but the associated values need not be. If a `Map` contains unique keys and unique values, it's said to implement a **one-to-one mapping**. If only the keys are unique, the `Map` is said to implement a **many-to-one mapping**—many keys can map to one value.

`Map`s differ from `Set`s in that `Map`s contain keys and values, whereas `Set`s contain only values. Two classes that implement interface `Map` are `HashMap` and `TreeMap`:

- **`HashMap`s** store elements in hash tables, and
- **`TreeMap`s** store elements in trees.

This section uses a `HashMap` to store key–value pairs. **Interface `SortedMap`** extends `Map` and maintains its keys in sorted order—either the elements' natural order or an order specified by a `Comparator`. Class `TreeMap` implements `SortedMap`.

Figure 12.15 uses a `HashMap` to count the number of occurrences

of each word in a string. Line 12 creates an empty `HashMap` with a default initial capacity (16 elements) and a default load factor (0.75)—these defaults are built into the implementation of `HashMap`. When the number of occupied slots in the `HashMap` exceeds the capacity times the load factor, the capacity is doubled automatically.

Click here to view code image

```
1  // Fig. 12.15: WordTypeCount.java
2  // Program counts the number of occurrences of each word in a String.
3  import java.util.Map;
4  import java.util.HashMap;
5  import java.util.Set;
6  import java.util.TreeSet;
7  import java.util.Scanner;
8
9  public class WordTypeCount {
10    public static void main(String[] args) {
11       // create HashMap to store String keys and Integer counts
12       Map<String, Integer> myMap = new HashMap<>();
13
14       createMap(myMap); // create map based on user input
15       displayMap(myMap); // display map content
16    }
17
18    // create map from user input
19    private static void createMap(Map<String, Integer> map) {
20       var scanner = new Scanner(System.in); // create scanner
21       System.out.println("Enter a string:"); // prompt for user input
22       String input = scanner.nextLine();
23
24       // tokenize the input
25       String[] tokens = input.split(" ");
26
27       // processing input text
28       for (String token : tokens) {
29          String word = token.toLowerCase(); // get lowercase word
30
31          // if the map contains the word
32          if (map.containsKey(word)) { // is word in map?
33             int count = map.get(word); // get current count
34             map.put(word, count + 1); // increment count
35          }
36          else {
37             map.put(word, 1); // add new word with a count of 1 to map
38          }
39       }
40    }
```

```
41
42      // display map content
43      private static void displayMap(Map<String, Integer> map) {
44          Set<String> keys = map.keySet(); // get a Set of the keys
45
46          // sort keys
47          var sortedKeys = new TreeSet<String>(keys);
48
49          System.out.printf("%nMap contains:%nKey\t\tValue%n");
50
51          // generate output for each key in map
52          for (String key : sortedKeys) {
53             System.out.printf("%-10s%10s%n", key, map.get(key));
54          }
55
56          System.out.printf(
57             "%nsize: %d%nisEmpty: %b%n", map.size(), map.isEmpty());
58       }
59    }
```

```
Enter a string:
this is a sample sentence with several words this is another sample
sentence with several different words

Map contains:
Key              Value
a                    1
another              1
different            1
is                   2
sample               2
sentence             2
several              2
this                 2
with                 2
words                2

size: 10
isEmpty: false
```

Fig. 12.15 | Program counts the number of occurrences of each word in a `String`.

`HashMap` is a generic class that takes two type arguments—the type of key (i.e., `String`) and the type of value (i.e., `Integer`). Recall that the type arguments passed to a generic class must be reference types, so the second type argument is `Integer`, not `int`.

Method `createMap`

Line 14 calls method `createMap` (lines 19–40), which uses a `Map` to store the number of occurrences of each word in the sentence. Line 22 obtains the user input, and line 25 tokenizes it. For each token, lines 28–39 convert the token to lowercase letters (line 29) for case-insensitive word counting, then call **`Map` method `containsKey`** (line 32) to determine whether the word is in the map (and thus has occurred previously in the string):

- If the word exists in the map, line 33 uses **`Map` method `get`** to obtain the key's associated value (the count) in the map. Line 34 increments that value and uses `put` to replace the key's associated value with a boxed `Integer` version of the new count. Method `put` returns the key's prior associated value, or `null` if the key was not in the map.
- If the `Map` does not contain the word, line 37 uses **`Map` method `put`** to create a new entry, with the word as the key and an `Integer` object containing `1` as the value (`1` is boxed).

Keys in `Map`s should be immutable objects. The key determines where a corresponding value is placed. Modifying a key after using it to insert a key–value pair could prevent a program from locating the key's corresponding value later. In this chapter's examples, we use `String`s as keys because they're immutable.

Method `displayMap`

Method `displayMap` (lines 43–58) displays all the entries in the map. It uses **`HashMap` method `keySet`** (line 44) to get a set of the keys. The keys have type `String` in the `map`, so `keySet` returns a `Set<String>`. Line 47 creates a `TreeSet` of the keys to store them in sorted order. Lines 52–54 access the map's key–value pairs. Line 53 displays each key and its value using format specifier `%-10s` to left align each key and format specifier `%10s` to right align each value. The keys are displayed in ascending order. Line 57 calls **`Map` method `size`** to get the number of key–value pairs in the `Map`, and calls **`Map` method `isEmpty`**, which returns a `boolean` indicating whether the `Map` is empty.

Generative AI

1 Prompt genAIs, asking what the default initial capacity and load

factor are for a Java `HashMap` and why these particular values were chosen.

2 Prompt genAIs, asking why a Java `HashMap`'s capacity is doubled when the `HashMap`'s number of occupied slots becomes greater than the capacity times the load factor.

3 Prompt genAIs, asking why a Java `Map`'s keys should be immutable.

12.11 Convenience Factory Methods for Creating Immutable Collections

Interfaces `List`, `Set` and `Map` contain `static` **convenience factory methods** for creating small immutable collections. The word "factory" indicates that these methods create objects. They're convenient because you simply pass the elements as arguments to a convenience factory method—it creates the collection and adds elements to it for you. Calling `List`, `Set` or `Map` methods that attempt to add or remove elements to these immutable collections results in `UnsupportedOperationExceptions`. Though the collections are immutable, they may contain mutable objects. The convenience factory methods throw `NullPointerExceptions` if they receive any `null` arguments.

Chapter 14 and 18 explain how using lambdas and streams with immutable entities can help you create "parallelizable" code that will run more efficiently on today's multi-core architectures. Figure 12.16 demonstrates these convenience factory methods for a `List`, a `Set` and two `Map`s.

Click here to view code image

```
1  // Fig. 12.16: FactoryMethods.java
2  // Convenience factory methods for creating immutable collections.
3  import java.util.List;
4  import java.util.Map;
5  import java.util.Set;
6
7  public class FactoryMethods {
8     public static void main(String[] args) {
9        // create a List
10       List<String> colorList = List.of("red", "orange", "yellow",
11          "green", "blue", "indigo", "violet");
```

```
        System.out.printf("colorList: %s%n%n", colorList);

        // create a Set
        Set<String> colorSet = Set.of("red", "orange", "yellow",
           "green", "blue", "indigo", "violet");
        System.out.printf("colorSet: %s%n%n", colorSet);

        // create a Map using method "of"
        Map<String, Integer> dayMap = Map.of("Monday", 1, "Tuesday", 2,
           "Wednesday", 3, "Thursday", 4, "Friday", 5, "Saturday", 6,
           "Sunday", 7);
        System.out.printf("dayMap: %s%n%n", dayMap);

        // create a Map using method "ofEntries" for more than 10 pairs
        Map<String, Integer> daysPerMonthMap = Map.ofEntries(
           Map.entry("January", 31),
           Map.entry("February", 28),
           Map.entry("March", 31),
           Map.entry("April", 30),
           Map.entry("May", 31),
           Map.entry("June", 30),
           Map.entry("July", 31),
           Map.entry("August", 31),
           Map.entry("September", 30),
           Map.entry("October", 31),
           Map.entry("November", 30),
           Map.entry("December", 31)
        );
        System.out.printf("monthMap: %s%n", daysPerMonthMap);
     }
  }
```

```
colorList: [red, orange, yellow, green, blue, indigo, violet]

colorSet: [yellow, green, red, blue, violet, indigo, orange]

dayMap: {Tuesday=2, Wednesday=3, Friday=5, Thursday=4, Saturday=6, Monday=1,
Sunday=7}

monthMap: {April=30, February=28, September=30, July=31, October=31,
November=30, December=31, March=31, January=31, June=30, May=31, August=31}
```

```
colorList: [red, orange, yellow, green, blue, indigo, violet]

colorSet: [violet, yellow, orange, green, blue, red, indigo]

dayMap: {Saturday=6, Tuesday=2, Wednesday=3, Sunday=7, Monday=1, Thursday=4,
Friday=5}
```

```
monthMap: {February=28, August=31, July=31, November=30, April=30, May=31,
December=31, September=30, January=31, March=31, June=30, October=31}
```

Fig. 12.16 | Convenience factory methods for creating immutable collections.

`List.of` Convenience Factory Method

Lines 10–11 use the **`List.of` convenience factory method** to create an immutable `List<String>`. Method `List.of` is overloaded for `List`s of 0–10 elements and provides an overload that receives any number of elements. The elements are guaranteed to be in the same order as `List.of`'s arguments. Line 12 displays the `List`'s `String` representation.

Convenience Factory Methods Are Optimized for Small Immutable Collections

According to JEP 269, which introduced convenience factory methods, these methods are overloaded to create collections of 0–10 elements because those "will be sufficient to cover the majority of use cases" in which programs require small immutable collections.[3] These overloads eliminate the extra overhead of processing variable-length argument lists, improving program performance. Similar overloads are provided for `Set`s of 0–10 elements and `Map`s of 0–10 key–value pairs. Also, the `List`, `Set` and `Map` interfaces each provide an overloaded factory method for collections of more than 10 elements or key–value pairs.

3. "JEP 269: Convenience Factory Methods for Collections." Accessed September 24, 2024. https://openjdk.org/jeps/269.

`Set.of` Convenience Factory Method

Lines 15–16 use the **`Set.of` convenience factory method** to create an immutable `Set<String>`. As with `List.of`, `Set.of` is overloaded for `Set`s of 0–10 elements and provides an overload for any number of elements. `Set.of` throws an `IllegalArgumentException` if any of its arguments are duplicates. Line 17 displays the `Set`'s `String` representation. Note that the order of the `Set`'s elements is different in each of the two sample outputs. **The iteration order is unspecified for the immutable `Set`s returned by the convenience factory methods**[4]—you'll ask why in Generative AI question 3.

4. “Interface `Set<E>`—Unmodifiable Sets.” Accessed September 24, 2024. https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/util/Set.html#unmodifiable.

`Map.of` Convenience Factory Method

Lines 20–22 use the **`Map.of` convenience factory method** to create an immutable `Map<String, Integer>`. As with `List` and `Set`, `Map.of` is overloaded for `Map`s of 0–10 key–value pairs—for larger `Map`s, `Map` provides method `ofEntries`, which we discuss momentarily. Each pair of arguments (for example, `"Monday"` and `1` in line 20) represents a key–value pair. Line 23 displays the `Map`’s `String` representation. **The iteration order is unspecified for the keys in `Map`s returned by the convenience factory methods**[5]—as the outputs indicate, the order can change between program executions. Methods `Map.of` and `Map.ofEntries` throw `IllegalArgumentException`s if any of the keys are duplicates.

5. “Interface `Map<K,V>`—Unmodifiable Maps.” Accessed September 21, 2024. https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/util/Map.html#unmodifiable.

`Map.ofEntries` Convenience Factory Method

Lines 26–39 use the **`Map.ofEntries` convenience factory method** to create an immutable `Map<String, Integer>`. Each of this method’s variable number of arguments is a **`Map.Entry`** object returned by a call to **`Map`’s `static` method `entry`**—a `Map.Entry` stores one key–value pair. Line 40 displays the `String` representation of the `Map`’s contents. Again, the outputs confirm that the iteration order of a `Map`’s keys can change between program executions.

Generative AI

1 Prompt genAIs, asking for use-cases for the immutable collections returned by Java’s convenience factory methods.

2 Prompt genAIs, asking why Java’s convenience factory methods are optimized primarily for small collections.

3 Prompt genAIs, asking why the immutable `Set`s and `Map`s returned by Java’s collection factory methods can have different iteration orders between program executions.

12.12 Concurrent Collections

Chapter 18 presents concurrency and parallelism. The collections in the collections framework are unsynchronized by default, so they can operate efficiently when concurrency and parallelism are not required. However, concurrent access to a collection by multiple threads could cause indeterminate results or fatal errors—as we demonstrate in Chapter 18. To fix such issues, we'll introduce the `java.util.concurrent` package's `ArrayBlockingQueue` class—one of many robust concurrent collections meant for use in applications that take advantage of today's multi-core processors to improve performance.

12.13 Wrap-Up

This chapter introduced the Java collections framework. You learned the Java collection framework's `Collection` interface hierarchy and how to use these interfaces to program with collections polymorphically. You used classes `ArrayList` and `LinkedList`, which both implement the `List` interface. You used several predefined `Collections` class methods to perform common collection manipulations.

We presented Java's `Queue` interface and `PriorityQueue` implementation to manipulate a queue. We used the `Set` interface and class `HashSet` to manipulate an unordered collection of unique values, and the `SortedSet` interface and class `TreeSet` to manipulate a sorted collection of unique values. Next, we presented Java's interfaces and classes for manipulating collections of key–value pairs—`Map`, `SortedMap`, `HashMap` and `TreeMap`. Finally, we introduced the convenience factory methods for creating immutable `List`s, `Set`s and `Map`s. For additional information on the Java Collections Framework, visit

Click here to view code image

```
https://docs.oracle.com/javase/tutorial/collections/index.html
```

and

Click here to view code image

```
https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/
   util/doc-files/coll-index.html.
```

Chapter 14, Functional Programming with Lambdas & Streams, introduces capabilities that will enable you to reimplement many of this chapter's examples more concisely and elegantly, and in a way that makes them easier to parallelize to improve performance on multi-core systems. Chapter 18, Concurrency: Platform Threads to Virtual Threads, discusses how to improve performance on multi-core systems using Java's concurrent collections and parallel-stream operations. See the book's web page at `https://deitel.com/javafp5` for a discussion of Java's sequenced collections, which were introduced in Java 21. In the next chapter, we continue our generics discussion by declaring a custom generic method and a custom generic class. We also discuss how the Java compiler handles generics.

Special Section: Generative AI Exercises

For each of the following questions, try multiple genAIs to determine which performs best. Always verify the results. Keep in mind that they sometimes make mistakes and even hallucinate, creating unreliable information.

12.1 ***(Generative AI: ArrayList vs. LinkedList)*** Prompt genAIs, asking them to suggest when to use `ArrayList` vs. `LinkedList`.

12.2 ***(Generative AI: HashSet vs. SortedSet)*** Prompt genAIs, asking them to suggest when to use `HashSet` vs. `SortedSet`.

12.3 ***(Generative AI: HashMap vs. SortedMap)*** Prompt genAIs, asking them to suggest when to use `HashMap` vs. `SortedMap`.

12.4 ***(Generative AI: Collections Optimized for Primitive Types)*** Java's generic collections can store only reference types, so primitive values must be boxed to store them in a collection. Prompt genAIs, asking them to overview and provide examples of using open-source Java libraries that manipulate primitive values without using boxing/unboxing.

12.5 ***(Generative AI: Collections Methods for Creating Singletons)*** Prompt genAIs, asking for real-world uses of `Collections` methods `singletonList`, `singleton`, and `singletonMap` methods for creating single-element collections.

12.6 ***(Generative AI: Choosing the Right Initial Size of a Collection)*** Prompt genAIs, asking how to choose an appropriate initial size for a Java collection to avoid frequent resizing.

Generic Classes and

Methods: A Deeper Look

Objectives

In this chapter, you'll:

- Create generic methods that perform identical tasks on arguments of different types.
- Restrict type parameters to ensure objects meet a generic method's requirements.
- Create a generic `Stack` class that can store objects of any reference type.
- Learn how Java uses type erasure to replace generic types with actual types during compile-time translation of generic methods and classes.
- Learn how to overload generic methods with non-generic and/ or generic methods.
- Use wildcards to specify that a type used with generics must be a subtype or supertype of a parameterized type.

Outline

13.1 Introduction

13.2 Motivation for Generic Methods

13.3 Generic Methods: Implementation and Compile-Time Translation

13.3.1 Generic Method `printArray`

13.3.2 Type Parameter Section of a Generic Method

13.3.3 Testing the Generic Method

13.3.4 Erasure at Compilation Time

13.4 Additional Compile-Time Translation Issues: Methods That Use a Type Parameter as the Return Type

13.5 Overloading Generic Methods

13.6 Generic Classes

13.6.1 Parameterized Types
13.6.2 Implementing a Generic `Stack` Class
13.6.3 Testing the Generic `Stack` Class
13.6.4 Refactoring the Test Program to Manipulate `Stack<E>` Objects with Generic Methods
13.7 Wildcards in Methods That Accept Type Parameters
13.7.1 Summing a `List<Number>`
13.7.2 Reimplementing Method `sum` with a Wildcard Type Argument in Its Parameter
13.8 Wrap-Up
Special Section: Generative AI Exercises

13.1 Introduction

You've used existing generic methods and classes in Chapters 6 and 12. In this chapter, you'll learn how to write your own.

It would be nice if we could write a single `sort` method to sort the elements in an `Integer` array, a `String` array or an array of any type that supports ordering (i.e., its elements can be compared). It would also be nice if we could write a single `Stack` class that could be used as a `Stack` of integers, a `Stack` of floating-point numbers, a `Stack` of `String`s or a `Stack` of any other type. It would be even nicer if we could detect type mismatches at compile time—known as **compile-time type safety**. For example, if a `Stack` should store only integers, an attempt to push a `String` onto that `Stack` should issue a compilation error. This chapter discusses **generics**—specifically **generic methods** and **generic classes**—which provide the means to create the type-safe general models mentioned above.

Generative AI

1 Prompt genAIs to list several Java API commonly used generic classes that are not collections.

13.2 Motivation for Generic Methods

Overloaded methods typically perform similar operations on different data types. To motivate generic methods, Fig. 13.1 uses overloaded `printArray` methods (lines 17–24 and 27–34) that display the `String` representations of the elements in an `Integer` array and a `Double` array, respectively. For this example, we could have used arrays of primitive types `int` and `double`. However, we're using arrays of the type-wrapper classes `Integer` and a `Double` to set up our generic method example—only reference types can be manipulated in generic methods and classes.[1]

1. In a future Java version, language enhancements being developed in Project Valhalla will enable Java generics to support primitive types directly. For more information visit https://openjdk.org/projects/valhalla/.

Click here to view code image

```
1  // Fig. 13.1: OverloadedMethods.java
2  // Printing array elements using overloaded methods.
3
4  public class OverloadedMethods {
5     public static void main(String[] args) {
6        // create arrays of Integers and Doubles
7        Integer[] integerArray = {1, 2, 3, 4, 5, 6};
8        Double[] doubleArray = {1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7};
9
10       System.out.printf("Array integerArray contains: ");
11       printArray(integerArray); // pass an Integer array
12       System.out.printf("Array doubleArray contains: ");
13       printArray(doubleArray); // pass a Double array
14    }
15
16    // method printArray to print Integer array
17    public static void printArray(Integer[] inputArray) {
18       // display array elements
19       for (Integer element : inputArray) {
20          System.out.printf("%s ", element);
21       }
22
23       System.out.println();
24    }
25
26    // method printArray to print Double array
27    public static void printArray(Double[] inputArray) {
28       // display array elements
29       for (Double element : inputArray) {
30          System.out.printf("%s ", element);
```

```
31          }
32
33          System.out.println();
34      }
35  }
```

```
Array integerArray contains: 1 2 3 4 5 6
Array doubleArray contains: 1.1 2.2 3.3 4.4 5.5 6.6 7.7
```

Fig. 13.1 | Printing array elements using overloaded methods.

The program declares and initializes two arrays:

- six-element `Integer` array `integerArray` (line 7), and
- seven-element `Double` array `doubleArray` (line 8).

Lines 11 and 13 each call `printArray` to display each array's contents.

When the compiler encounters a method call, it attempts to locate a method declaration with the same name and parameters that match the call's argument type(s). In this example, each `printArray` call matches one `printArray` declaration:

- Line 11 calls `printArray` with the argument `integerArray`. The compiler determines the argument's type (`Integer[]`) and attempts to locate a `printArray` method with an `Integer[]` parameter (lines 17–24), then sets up a call to that method.
- Similarly, when the compiler encounters the call at line 13, it determines the argument's type (`Double[]`) and attempts to locate a `printArray` method with a `Double[]` parameter (lines 27–34), then sets up a call to that method.

Common Features in the Overloaded `printArray` Methods

Study each `printArray` method. The element type appears in each method's header (lines 17 and 27) and `for`-statement header (lines 19 and 29). If we were to replace the element types in each method with a generic name—such as `T`—then both methods would appear as follows:

Click here to view code image

```
1  public static void printArray(T[] inputArray) {
2     // display array elements
3     for (T element : inputArray)  {
4        System.out.printf("%s ", element);
5     }
6
7     System.out.println();
8  }
```

If we can replace the array element type in each method with a single generic type, we should be able to declare one `printArray` method that can display the `String` representations of any array's elements. **The method above will not compile**. This code highlights that Fig. 13.1's `printArray` methods are identical other than the types they process. Section 13.3 presents a correctly declared generic `printArray` method.

Generative AI

1 Prompt genAIs for an introduction to Project Valhalla and how it will affect boxing and unboxing in Java.

13.3 Generic Methods: Implementation and Compile-Time Translation

If the operations performed by overloaded methods are identical for each argument type, the overloaded methods can be coded more conveniently as a generic method that can be called with arguments of different types. Based on the argument types passed to the generic method, the compiler handles each method call appropriately. At compilation time, the compiler ensures the type safety of your code, preventing many runtime errors.

13.3.1 Generic Method `printArray`

Figure 13.2 reimplements Fig. 13.1 using a generic `printArray` method (lines 17–24 of Fig. 13.2). The `printArray` calls in lines 11 and 13 are identical to those in Fig. 13.1, and the two programs' outputs are identical. This demonstrates the expressive power of generics.

Click here to view code image

```java
// Fig. 13.2: GenericMethodTest.java
// Printing array elements using generic method printArray.

public class GenericMethodTest {
   public static void main(String[] args) {
      // create arrays of Integer, Double and Character
      Integer[] integerArray = {1, 2, 3, 4, 5, 6};
      Double[] doubleArray = {1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7};

      System.out.printf("Array integerArray contains: ");
      printArray(integerArray); // pass an Integer array
      System.out.printf("Array doubleArray contains: ");
      printArray(doubleArray); // pass a Double array
   }

   // generic method printArray
   public static <T> void printArray(T[] inputArray) {
      // display array elements
      for (T element : inputArray) {
         System.out.printf("%s ", element);
      }

      System.out.println();
   }
}
```

```
Array integerArray contains: 1 2 3 4 5 6
Array doubleArray contains: 1.1 2.2 3.3 4.4 5.5 6.6 7.7
```

Fig. 13.2 | Printing array elements using generic method `printArray`.

13.3.2 Type Parameter Section of a Generic Method

Lines 17–24 eliminate Fig. 13.1's overloaded methods by creating a generic `printArray` method that can output the `String` representations of any array of objects. All generic method declarations have a **type-parameter section** delimited by **angle brackets**—in this case, `<T>` in line 17—which precedes the method's return type. A type-parameter section contains one or more **type parameters** (`T` in line 17) separated by commas. A type parameter—also known as a **type variable**—is an identifier that specifies a generic type name. The type parameters can be used to declare the return type, parameters and local variables in a generic

method declaration. They act as placeholders for the argument types passed to the generic method, which are known as **actual type arguments**. Type parameters can represent only reference types—not primitive types (like `int` and `double`).

A generic method's body is declared like any other method. Type-parameter names throughout the method declaration must match names declared in the type-parameter section. For example, line 19 declares `element` as type `T`—the type parameter we declared in line 17. A type parameter may be declared only once in the type-parameter section but can appear many times throughout the method's header and body. In the following method header, the type-parameter name `T` appears twice in the method's parameter list and is also used as the method's return type:

Click here to view code image

```
public static <T> T maximum(T value1, T value2)
```

Type-parameter names need not be unique among different generic methods. In generic method `printArray`, `T` appears in the two locations where Fig. 13.1's overloaded `printArray` methods specified `Integer` or `Double` as the array's element type. The other code in `printArray` is identical to the overloaded versions in Fig. 13.1.

Common Type Parameter Names

We used the generic type-parameter name `T` in method `printArray`. Several letters are commonly used for generic type names, but any valid identifier is allowed:[2]

2. "Generic Types." Accessed September 4, 2024. https://docs.oracle.com/javase/tutorial/java/generics/types.html.

- `T` for a "type" (and `S`, `U`, `V` if additional generic types are required).
- `E` for an "element" type in a collection.
- `K` for a "key" type in a `Map`.
- `V` for a "value" type in a `Map`.

13.3.3 Testing the Generic Method

Figure 13.2 declares and initializes a six-element `Integer` array `integerArray` (line 7) and a seven-element `Double` array

`doubleArray` (line 8). Lines 11 and 13 output each array by calling `printArray`. When the compiler encounters line 11, it determines `integerArray`'s type (`Integer[]`) and attempts to locate a method named `printArray` that specifies an `Integer[]` parameter. There's no such method in this example, so the compiler determines whether there's a generic `printArray` method with an array parameter that represents the element type specified with a type parameter. The compiler determines that `printArray` (lines 17–24) is a match and sets up a call to the method. The same process is repeated for line 13's call to method `printArray`.

Compilation Errors If No Match Is Found

A compilation error occurs if the compiler cannot match a method call to a non-generic or generic method declaration. A compilation error also occurs if the compiler finds two or more methods that satisfy a given method call. For the complete details of resolving calls to overloaded and generic methods, visit

Click here to view code image

```
https://docs.oracle.com/javase/specs/jls/se22/html/jls-15.html#jls-
   15.12
```

Verifying the Method's Body

The compiler also ensures that the operations in `printArray`'s body can be applied to elements of the type stored in its array argument. The only operation performed on the array elements in this example is to output their `String` representations. Line 20 performs an implicit `toString` call on every `element`. To work with generics, every element must refer to an object and all objects have a `toString` method, so the compiler is satisfied that line 20 performs a valid operation. `Integer`'s and `Double`'s `toString` methods return the `String` representations of their underlying `int` or `double` values.

13.3.4 Erasure at Compilation Time

When the compiler translates the generic method `printArray` into Java bytecodes, it removes the type-parameter section and replaces each occurrence of the type parameter with actual types—a process known as **type erasure**. By default, all generic types are replaced with type `Object`. So, the compiled version of `printArray` is:

Click here to view code image

```
1  public static void printArray(Object[] inputArray) {
2     // display array elements
3     for (Object element : inputArray) {
4        System.out.printf("%s ", element);
5     }
6
7     System.out.println();
8  }
```

One copy of this code is used for all `printArray` calls in the example. This is quite different from similar mechanisms in other programming languages, such as C++'s templates, in which the compiler generates and compiles a separate copy of the source code for every type passed as an argument to the method. Section 13.4 discusses the translation and compilation of generics, which is more involved than what we've discussed so far.

Figure 13.2's generic `printArray` method eliminated the need for the overloaded methods of Fig. 13.1 and created a reusable method that can output the `String` representations of any array of objects. However, we could have simply declared `printArray` to receive an `Object[]` parameter. The benefits of generic methods become more apparent when you restrict the types they can process, as we do in Section 13.4

Generative AI

1 Prompt genAIs to give several examples of how Java's compile-time type safety with generics can eliminate potential runtime errors.

13.4 Additional Compile-Time Translation Issues: Methods That Use a Type Parameter as the Return Type

Let's consider a generic method that uses type parameters in the return type and parameter list (Fig. 13.3). The program uses a

generic `maximum` method (similar to `maximum` in Fig. 5.1) to determine and return the largest of its three arguments, which must all have the same type.

Click here to view code image

```
// Fig. 13.3: MaximumTest.java
// Generic method maximum returns the largest of three objects.

public class MaximumTest {
   public static void main(String[] args) {
      System.out.printf("Maximum of %d, %d and %d is %d%n", 3, 4, 5,
         maximum(3, 4, 5));
      System.out.printf("Maximum of %.1f, %.1f and %.1f is %.1f%n",
         6.6, 8.8, 7.7, maximum(6.6, 8.8, 7.7));
      System.out.printf("Maximum of %s, %s and %s is %s%n", "pear",
         "apple", "orange", maximum("pear", "apple", "orange"));
   }

   // determines the largest of three objects
   public static <T extends Comparable<T>> T maximum(T x, T y, T z) {
      T max = x; // assume x is initially the largest

      if (y.compareTo(max) > 0) {
         max = y; // y is the largest so far
      }

      if (z.compareTo(max) > 0) {
         max = z; // z is the largest
      }

      return max; // returns the largest object
   }
}
```

```
Maximum of 3, 4 and 5 is 5
Maximum of 6.6, 8.8 and 7.7 is 8.8
Maximum of pear, apple and orange is pear
```

Fig. 13.3 | Generic method `maximum` returns the largest of three objects.

Java Objects Cannot Be Compared with Relational Operators

Relational operators like `>` cannot be used with reference types.[3] However, you can compare two objects of a class if the class

implements the generic interface `Comparable<T>` (package `java.lang`). The type-wrapper classes (like `Integer` and `Double`) implement this interface. **Generic interfaces** enable you to specify a set of related types with a single interface declaration. `Comparable<T>` objects have a **`compareTo` method**. For example, you can compare two `BigInteger` objects (Section 3.12), `integer1` and `integer2`, with the expression:

3. This works for type wrapper objects due to unboxing, but not other kinds of objects.

```
integer1.compareTo(integer2)
```

because `BigInteger` implements `Comparable<BigInteger>`. Classes `String` (Chapter 7) and `BigDecimal` (Section 4.11) also implement `Comparable<T>`.

A class that implements `Comparable<T>` must define the method `compareTo` such that it compares the contents of two objects of the class and returns the comparison results. The method call `object1.compareTo(object2)` must return

- `0` if the `object1` and `object2` are equal,
- a negative integer if `object1` is less than `object2` or
- a positive integer if `object1` is greater than `object2`.

For example, class `Integer`'s `compareTo` method compares the `int` values stored in two `Integer` objects. In this example, we'll use `compareTo` in method `maximum` to help determine the largest of its three arguments.

Generic Method `maximum` and Specifying a Type Parameter's Upper Bound

Generic method `maximum` (lines 15–27) uses type parameter `T` as

- its return type (line 15),
- the type of its parameters `x`, `y` and `z` (line 15), and
- the type of local variable `max` (line 16).

The type-parameter section specifies

```
T extends Comparable<T>
```

indicating only objects of classes that implement interface

`Comparable<T>` can be used with method `maximum`. `Comparable<T>` is known as the **type parameter's upper bound**. By default, a type parameter's upper bound is `Object`, meaning any reference type can be used. Type-parameter declarations that bound the type parameter use keyword `extends` regardless of whether the type parameter `extends` a class or `implements` an interface. The upper bound may be a comma-separated list that contains zero or one class and zero or more interfaces.

Method `maximum`'s type parameter is more restrictive than `printArray`'s in Fig. 13.2. The `Comparable<T>` restriction is important—only objects of classes that implement `Comparable<T>` are guaranteed to have the `compareTo` method used in lines 18 and 22. Without the upper bound for `maximum`'s type parameter, the default upper bound would be `Object` and the compiler would generate error messages like the following on each call to `compareTo`:

Click here to view code image

```
MaximumTest.java:18: error: cannot find symbol
      if (y.compareTo(max) > 0) {
           ^
   symbol: method compareTo(T)
   location: variable y of type T
   where T is a type-variable:
     T extends Object declared in method <T>maximum(T,T,T)
```

Logic of Method `maximum`

Method `maximum` uses the same algorithm as Fig. 5.1 to determine the largest of its three arguments:

- The method assumes that its first argument (`x`) is the largest and assigns it to `max` (line 16 of Fig. 13.3).
- The `if` statement in lines 18–20 uses the expression `y.compareTo(max)` to determines `y`'s relationship to `max`. If `compareTo`'s return value is greater than `0`, `y` is greater than `max`, so line 19 assigns `y` to `max`.
- Similarly, the `if` statement at lines 22–24 determines whether `z` is greater than `max` and, if so, assigns `z` to `max`.
- Then, line 26 returns the `max` value to the caller.

Calling Method `maximum`

In `main`, line 7 calls `maximum` with the integers `3`, `4` and `5`. When the compiler encounters this call, it first looks for a `maximum` method that takes three arguments of type `int`. There's no such method, so the compiler looks for a generic method that takes three arguments of the same type and finds the method `maximum`. Recall that a generic method's arguments must have reference types. So the compiler boxes the `int` arguments as `Integer` objects and passes them to `maximum`. The `Integer` class implements `Comparable<Integer>`, so `Integer`s are valid arguments to `maximum`. When the `maximum` method returns the largest `Integer`, we output it with the `%d` format specifier, which expects an `int` primitive-type value, so the JVM unboxes the `int` from the `Integer`.

This process repeats for line 9's `maximum` call. Each `double` is boxed as a `Double` object and passed to `maximum`. `Double` implements `Comparable<Double>`, so `Double`s are valid arguments to `maximum`. The `Double` returned by `maximum` is unboxed and displayed with the format specifier `%.1f`.

Line 11's `maximum` call receives three `String`s. This is allowed because class `String` implements `Comparable<String>`. We intentionally placed the largest value in a different position in each `maximum` call (lines 7, 9 and 11) to demonstrate that `maximum` always finds the largest value, regardless of its position in the argument list.

Type Erasure and the Upper Bound of a Type Parameter

When the compiler translates `maximum` into bytecodes, it uses type erasure to replace the type parameters with actual types. We discussed previously that, by default, the compiler replaces generic types with `Object`. However, for a type parameter with an upper bound, each occurrence is replaced with its upper bound from the method's the type-parameter section. The following code simulates the type erasure of method `maximum`'s generic types by removing the type-parameter section and replacing `T` with the upper bound `Comparable` throughout the method declaration—the erasure of `Comparable<T>` is `Comparable`:

Click here to view code image

```
1  public static Comparable maximum(Comparable x, Comparable y,
2     Comparable z) {
3
4     Comparable max = x; // assume x is initially the largest
```

```
5
6       if (y.compareTo(max) > 0) {
7          max = y; // y is the largest so far
8       }
9
10      if (z.compareTo(max) > 0) {
11         max = z; // z is the largest
12      }
13
14      return max; // returns the largest object
15   }
```

After type erasure, method `maximum` specifies that it returns type `Comparable`. However, the calling method (`main`) does not expect to receive a `Comparable`. It expects to receive an object of the same type it passed to `maximum` as an argument—an `Integer`, `Double` or `String`, in this example. When the compiler replaces the type-parameter information with the upper-bound type in the method declaration, it also inserts explicit cast operators in front of each method call to ensure that `maximum`'s return value has type the caller expects. So, the `maximum` calls in lines 7, 9 and 11 (Fig. 13.3) are rewritten by the compiler as:

Click here to view code image

```
(Integer) maximum(3, 4, 5)
(Double) maximum(6.6, 8.8, 7.7)
(String) maximum("pear", "apple", "orange")
```

The cast type is inferred from the method's argument types because the method declaration indicates that return type and the argument types match.

Generative AI

1 Prompt genAIs to define operator overloading and explain why Java's designers chose not to have the language support it.

2 Prompt genAIs for examples of commonly used Java objects that cannot be compared.

3 Prompt genAIs to rewrite the Java program in Fig. 13.3, similar to how the Java compiler performs type erasure on a generic method. Ask them to include the appropriate cast operations. Compare their results with the original code.

13.5 Overloading Generic Methods

A generic method may be overloaded like any other method. In addition, generic methods in a superclass can be overridden in a subclass with generic or non-generic methods.

A class can provide two or more generic methods with the same method name but different method parameters. For example, Fig. 13.2's generic method `printArray` could be overloaded with another `printArray` generic method that has the additional parameters `startIndex` and `endIndex` to specify a range of array elements to output.

A generic method can also be overloaded by a non-generic method. For example, Fig. 13.2's generic method `printArray` could be overloaded with a version specific to `String`s.

Generative AI

1 Prompt genAIs to provide and explain a Java code example in which a generic method is overloaded by generic and non-generic methods.

13.6 Generic Classes

A collection like a stack can be understood independently of its element type. Generic classes provide a means for describing the concept of a stack (or any other class) in a type-independent manner. We can then instantiate type-specific objects of the generic class, providing a nice opportunity for software reuse.

Once you have a generic class, you can use a simple, concise notation to indicate the type(s) that should be used in place of the class's type parameter(s). At compilation time, the compiler ensures your code's type safety and uses the type erasure techniques described in Sections 13.3–13.4 to enable your code to interact with the generic class.

13.6.1 Parameterized Types

One generic `Stack` class, for example, could be the basis for creating many `Stack` classes—for example, "`Stack` of `Double`," "`Stack` of `Integer`," "`Stack` of `Character`" and "`Stack` of `Employee`." These classes are known as **parameterized types** because they

accept one or more type parameters. Recall that type parameters represent only reference types, which means the `Stack` generic class cannot be instantiated with primitive types. However, we can instantiate a `Stack` that stores objects of Java's type-wrapper classes, using boxing to convert the primitive values to objects. Boxing occurs when a primitive-type value (e.g., `int`) is pushed onto a `Stack` that contains wrapper-class objects (e.g., `Integer`). Unboxing occurs when a type-wrapper object is popped off the `Stack` and used where a primitive-type value is expected.

13.6.2 Implementing a Generic `Stack` Class

Figure 13.4 declares a generic `Stack` class for demonstration purposes—the `java.util` package contains class `Stack<E>` capabilities, which you can use rather than creating a custom class.

Click here to view code image

```
1  // Fig. 13.4: Stack.java
2  // Stack generic class declaration.
3  import java.util.ArrayList;
4  import java.util.NoSuchElementException;
5
6  public class Stack<E> {
7     private final ArrayList<E> elements; // ArrayList stores stack elements
8
9     // no-argument constructor creates a stack of the default size
10    public Stack() {
11       this(10); // default stack size
12    }
13
14    // constructor creates a stack of the specified number of elements
15    public Stack(int capacity) {
16       int initCapacity = capacity > 0 ? capacity : 10; // validate
17       elements = new ArrayList<E>(initCapacity); // create ArrayList
18    }
19
20    // push element onto stack
21    public void push(E pushValue) {
22       elements.addLast(pushValue); // place pushValue on Stack
23    }
24
25    // return the top element if not empty; else throw exception
26    public E pop() {
27       if (elements.isEmpty()) { // if stack is empty
28          throw new NoSuchElementException("Stack is empty, cannot pop");
29       }
```

```
30
31          // remove and return top element of Stack
32          return elements.removeLast();
33       }
34    }
```

Fig. 13.4 | `Stack` generic class declaration.

A generic class declaration follows the class name with a type-parameter section (`<E>` in line 6). The type parameter `E` represents the `Stack`'s element type and is used to represent that type throughout the class declaration. As with generic methods, a generic class's type-parameter section can have multiple type parameters separated by commas.

Using an `ArrayList<E>` for the Stack's Elements

Class `Stack` declares its instance variable `elements` as an `ArrayList<E>` (line 7). This `ArrayList` will store the `Stack`'s elements—note that the `ArrayList` uses the same element type (`E`) as the `Stack`. As you know, an `ArrayList` is dynamically resizable, so our `Stack` class objects are as well.

`Stack` Constructors

The `Stack` class's no-argument constructor (lines 10–12) invokes the one-argument constructor (lines 15–18) to create a `Stack` in which the underlying `ArrayList` has a capacity of `10` elements. The one-argument constructor can also be called directly to create a `Stack` with a specified initial capacity. Line 16 validates the constructor's argument. Line 17 creates the `ArrayList` of the specified capacity (or 10 if the capacity was invalid).

`Stack` Methods `push` and `pop`

Method `push` (lines 21–23) uses `ArrayList` method `addLast` to append the pushed item to the end of the `ArrayList` `elements`. The last element in the `ArrayList` represents the stack's top. Method `pop` (lines 26–33) first determines whether the `Stack` is empty. If so, line 28 throws a `NoSuchElementException` (package `java.util`). Otherwise, line 32 returns the `Stack`'s top element by calling `ArrayList` method `removeLast`, which removes and returns the `ArrayList`'s last element.

`Stack`'s Type Parameter

As with generic methods, when a generic class is compiled, the compiler performs type erasure on the class's type parameters and replaces them with their upper bounds. Class `Stack` (Fig. 13.4) does not specify an upper bound for its type parameter, so `Object` is used by default. The scope of a generic class's type parameter is the entire class. However, type parameters cannot be used in a class's `static` variable declarations.

13.6.3 Testing the Generic `Stack` Class

Now, let's consider the program (Fig. 13.5) that uses the `Stack` generic class (Fig. 13.4). Lines 11–12 in Fig. 13.5 create and initialize variables of type `Stack<Double>` (pronounced "`Stack` of `Double`") and `Stack<Integer>` (pronounced "`Stack` of `Integer`"). `Double` and `Integer` are the type arguments for each `Stack`. The compiler uses them to perform type checking and insert cast operations as necessary as it performs type erasure. We'll discuss the cast operations in more detail shortly. Lines 11 instantiates `doubleStack` with a capacity of `5`, and line 12 instantiates `integerStack` with the default capacity of 10. Lines 15–16 and 19–20 call methods `testPushDouble` (lines 24–33), `testPopDouble` (lines 36–52), `testPushInteger` (lines 55–64) and `testPopInteger` (lines 67–83) to manipulate the two `Stack`s in this example.

Click here to view code image

```
1  // Fig. 13.5: StackTest.java
2  // Stack generic class test program.
3  import java.util.NoSuchElementException;
4
5  public class StackTest {
6     public static void main(String[] args) {
7        double[] doubleElements = {1.1, 2.2, 3.3, 4.4, 5.5};
8        int[] integerElements = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
9
10       // Create a Stack<Double> and a Stack<Integer>
11       var doubleStack = new Stack<Double>(5);
12       var integerStack = new Stack<Integer>();
13
14       // push elements of doubleElements onto doubleStack
15       testPushDouble(doubleStack, doubleElements);
16       testPopDouble(doubleStack); // pop from doubleStack
17
18       // push elements of integerElements onto integerStack
19       testPushInteger(integerStack, integerElements);
20       testPopInteger(integerStack); // pop from integerStack
```

```
21      }
22
23      // test push method with double stack
24      private static void testPushDouble(
25         Stack<Double> stack, double[] values) {
26         System.out.printf("%nPushing elements onto doubleStack%n   ");
27
28         // push elements to Stack
29         for (double value : values) {
30            System.out.printf("%.1f ", value);
31            stack.push(value); // push onto doubleStack
32         }
33      }
34
35      // test pop method with double stack
36      private static void testPopDouble(Stack<Double> stack) {
37         // pop elements from stack
38         try {
39            System.out.printf("%nPopping elements from doubleStack%n   ");
40            double popValue; // store element removed from stack
41
42            // remove all elements from Stack
43            while (true) {
44               popValue = stack.pop(); // pop from doubleStack
45               System.out.printf("%.1f ", popValue);
46            }
47         }
48         catch(NoSuchElementException noSuchElementException) {
49            System.err.println();
50            noSuchElementException.printStackTrace();
51         }
52      }
53
54      // test push method with integer stack
55      private static void testPushInteger(
56         Stack<Integer> stack, int[] values) {
57         System.out.printf("%nPushing elements onto integerStack%n   ");
58
59         // push elements to Stack
60         for (int value : values) {
61            System.out.printf("%d ", value);
62            stack.push(value); // push onto integerStack
63         }
64      }
65
66      // test pop method with integer stack
67      private static void testPopInteger(Stack<Integer> stack) {
68         // pop elements from stack
69         try {
70            System.out.printf("%nPopping elements from integerStack%n   ");
```

```
71            int popValue; // store element removed from stack
72
73            // remove all elements from Stack
74            while (true) {
75               popValue = stack.pop(); // pop from intStack
76               System.out.printf("%d ", popValue);
77            }
78         }
79         catch(NoSuchElementException noSuchElementException) {
80            System.err.println();
81            noSuchElementException.printStackTrace();
82         }
83      }
84   }
```

```
Pushing elements onto doubleStack
   1.1 2.2 3.3 4.4 5.5
Popping elements from doubleStack
   5.5 4.4 3.3 2.2 1.1
java.util.NoSuchElementException: Stack is empty, cannot pop
        at Stack.pop(Stack.java:28)
        at StackTest.testPopDouble(StackTest.java:44)
        at StackTest.main(StackTest.java:16)

Pushing elements onto integerStack
   1 2 3 4 5 6 7 8 9 10
Popping elements from integerStack
   10 9 8 7 6 5 4 3 2 1
java.util.NoSuchElementException: Stack is empty, cannot pop
        at Stack.pop(Stack.java:28)
        at StackTest.testPopInteger(StackTest.java:75)
        at StackTest.main(StackTest.java:20)
```

Fig. 13.5 | `Stack` generic class test program.

Methods `testPushDouble` and `testPopDouble`

Method `testPushDouble` (lines 24–33) invokes method `push` (line 31) to place the `double` values 1.1, 2.2, 3.3, 4.4 and 5.5 from array `doubleElements` onto `doubleStack`. Boxing occurs in line 31 when the program tries to push a primitive `double` value onto the `doubleStack`, which stores references to `Double` objects.

Method `testPopDouble` (lines 36–52) invokes `Stack` method `pop` (line 44) in an infinite loop (lines 43–46) to remove all the values from the stack. In the output, the values indeed `pop` in last-in, first-out (LIFO) order—the defining characteristic of stacks. When the loop attempts to `pop` a sixth value, the `doubleStack` is empty, so

`pop` throws a `NoSuchElementException`, which causes the program to proceed to the `catch` block (lines 48–51). The stack trace indicates the exception that occurred and that method `pop` generated the exception at line 28 of the file `Stack.java` (Fig. 13.4). The trace also indicates that `pop` was called by `StackTest` method `testPopDouble` at line 44 (Fig. 13.5) of `Stack-Test.java` and that method `testPopDouble` was called from method `main` at line 16 of `StackTest.java`. This information helps you determine the methods on the method-call stack when the exception occurred. Because the program catches the exception, it's considered to have been handled, and the program can continue executing. Terminating a loop via an exception is a poor programming practice.

Unboxing occurs in line 44 when the program assigns the `Double` object popped from the stack to a `double` primitive variable. Recall from Section 13.4 that the compiler inserts casts to ensure that the proper types are returned from generic methods. After type erasure, `Stack` method `pop` returns type `Object`, but the client code in `testPopDouble` expects to receive a `Double` when method `pop` returns. So, the compiler inserts a `Double` cast, as in

Click here to view code image

```
popValue = (Double) stack.pop();
```

The value assigned to `popValue` will be unboxed from the `Double` object returned by `pop`.

Methods `testPushInteger` and `testPopInteger`

Method `testPushInteger` (lines 55–64) invokes `Stack` method `push` to place the `int` values 1–10 from array `intElements` onto `integerStack`. Method `testPopInteger` (lines 67–83) invokes `Stack` method `pop` to remove values from `integerStack`. Once again, the values are popped in last-in, first-out order. During type erasure, the compiler recognizes that the client code in method `testPopInteger` expects to receive an `Integer` when method `pop` returns. So, the compiler inserts an `Integer` cast, as in

Click here to view code image

```
popValue = (Integer) stack.pop();
```

The value assigned to `popValue` will be unboxed from the `Integer` object returned by `pop`.

Generative AI

1 Prompt genAIs for insights on how Java generics provide opportunities for software reusability.
2 Prompt genAIs to list examples of Java generic classes that receive two or more type parameters.
3 Prompt genAIs, asking which Java generic collections can represent a stack and whether a particular collection is optimal.
4 Prompt genAIs, asking why implementing a stack using a Java `ArrayList` with the first element as the top is not optimal. In your prompt, ask the genAIs not to use Big O notation in their explanations (we discuss Big O notation in Chapter 22).
5 Prompt genAIs, asking why a Java generic class's type parameter(s) cannot be used in the class's `static` variable declarations.
6 Prompt genAIs, asking whether a `static` Java method may be a generic method and, if so, to give some examples of such methods from the Java API.

13.6.4 Refactoring the Test Program to Manipulate `Stack<E>` Objects with Generic Methods

In Fig. 13.5, you might have noticed that:

- the code in methods `testPushDouble` and `testPushInteger` is almost identical for pushing values onto a `Stack<Double>` and a `Stack<Integer>`, and
- the code in methods `testPopDouble` and `testPopInteger` is almost identical for popping values from a `Stack<Double>` and a `Stack<Integer>`.

This presents another opportunity to use generic methods. Figure 13.6 refactors the code using generic method `testPush` (lines 24–33) to perform the tasks of `testPushDouble` and `testPushInteger` in Fig. 13.5—that is, `push` values onto a `Stack<E>`. Similarly, generic method `testPop` (Fig. 13.6, lines 36–52) performs the tasks of `testPopDouble` and `test-PopInteger` in Fig. 13.5—that is, `pop` values off a `Stack<E>`. The output of Fig. 13.6 precisely matches that of Fig. 13.5.

Click here to view code image

```
1  // Fig. 13.6: StackTest2.java
2  // Passing generic Stack objects to generic methods.
3  import java.util.NoSuchElementException;
4
5  public class StackTest2 {
6     public static void main(String[] args) {
7        Double[] doubleElements = {1.1, 2.2, 3.3, 4.4, 5.5};
8        Integer[] integerElements = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
9
10       // Create a Stack<Double> and a Stack<Integer>
11       Stack<Double> doubleStack = new Stack<>(5);
12       Stack<Integer> integerStack = new Stack<>();
13
14       // push elements of doubleElements onto doubleStack
15       testPush("doubleStack", doubleStack, doubleElements);
16       testPop("doubleStack", doubleStack); // pop from doubleStack
17
18       // push elements of integerElements onto integerStack
19       testPush("integerStack", integerStack, integerElements);
20       testPop("integerStack", integerStack); // pop from integerStack
21    }
22
23    // generic method testPush pushes elements onto a Stack
24    public static <E> void testPush(String name, Stack<E> stack,
25       E[] elements) {
26       System.out.printf("%nPushing elements onto %s%n   ", name);
27
28       // push elements onto Stack
29       for (E element : elements) {
30          System.out.printf("%s ", element);
31          stack.push(element); // push element onto stack
32       }
33    }
34
35    // generic method testPop pops elements from a Stack
36    public static <E> void testPop(String name, Stack<E> stack) {
37       // pop elements from stack
38       try {
39          System.out.printf("%nPopping elements from %s%n   ", name);
40          E popValue; // store element removed from stack
41
42          // remove all elements from Stack
43          while (true) {
44             popValue = stack.pop();
45             System.out.printf("%s ", popValue);
46          }
47       }
48       catch(NoSuchElementException noSuchElementException) {
49          System.out.println();
50          noSuchElementException.printStackTrace();
```

```
51          }
52       }
53    }
```

```
Pushing elements onto doubleStack
   1.1 2.2 3.3 4.4 5.5
Popping elements from doubleStack
   5.5 4.4 3.3 2.2 1.1
java.util.NoSuchElementException: Stack is empty, cannot pop
        at Stack.pop(Stack.java:28)
        at StackTest2.testPop(StackTest2.java:44)
        at StackTest2.main(StackTest2.java:16)

Pushing elements onto integerStack
   1 2 3 4 5 6 7 8 9 10
Popping elements from integerStack
   10 9 8 7 6 5 4 3 2 1
java.util.NoSuchElementException: Stack is empty, cannot pop
        at Stack.pop(Stack.java:28)
        at StackTest2.testPop(StackTest2.java:44)
        at StackTest2.main(StackTest2.java:20)
```

Fig. 13.6 | Passing generic `Stack` objects to generic methods.

Lines 11–12 create the `Stack<Double>` and `Stack<Integer>` objects. Lines 15–16 and 19–20 invoke generic methods `testPush` and `testPop` to test the `Stack` objects. Generic method `testPush` (lines 24–33) uses type parameter `E` (line 24) to represent the `Stack`'s element type. The generic method takes three arguments:

- a `String` that represents the `Stack<E>` object's name for output purposes,
- a reference to a `Stack<E>` object and
- an array of type `E` representing the elements to `push` onto the `Stack`.

The compiler enforces consistency between the `Stack`'s element type and the elements that will be pushed onto the `Stack` when `push` is invoked—a key benefit of the generic method calls to `testPush`. Generic method `testPop` (lines 36–52) takes two arguments—a `String` that represents the `Stack<E>` object's name for output purposes and a reference to a `Stack<E>` object. It also displays the `String` representation of the popped element.

Generative AI

1 Prompt genAIs to refactor Fig. 13.5's code, replacing `testPushDouble` and `testPush-Integer` with a single generic method, and replacing `testPopDouble` and `testPopInteger` with a single generic method. Compare the generated results with the code in Fig. 13.6.

13.7 Wildcards in Methods That Accept Type Parameters

In this section, we introduce a powerful generics concept known as **wildcards**, which enable you to specify method parameters, return values, variables and fields that act as supertypes or subtypes of parameterized types.

13.7.1 Summing a `List<Number>`

First, let's motivate wildcards with an example (Fig. 13.7). Suppose you'd like to implement a generic method `sum` that totals the numeric values in a `List`. You'd begin with a `List` of numbers. As we've demonstrated previously, type parameters in generic classes must be reference types, so the numbers would be boxed in type-wrapper objects. We want to total all the numbers regardless of their type. So, we'll declare this example's `List` with the type argument `Number`, which is the superclass of the numeric type-wrapper classes, like `Integer` and `Double`. We'll also declare `sum`'s parameter as a `List<Number>` and total the parameter's elements.

Click here to view code image

```
1  // Fig. 13.7: TotalNumbers.java
2  // Totaling the numbers in a List<Number>.
3  import java.util.Arrays;
4  import java.util.List;
5
6  public class TotalNumbers {
7     public static void main(String[] args) {
8        // create List<Number> and add Integers and Doubles to it
9        Number[] numbers = {1, 2.4, 3, 4.1};
10       List<Number> numberList = Arrays.asList(numbers);
11
12       // display numbersList
13       System.out.printf("numberList contains: %s%n", numberList);
14
15       // total numbersList's elements then display the result
```

```
16          System.out.printf("Total of the elements in numberList: %.1f%n",
17             sum(numberList));
18       }
19
20       // calculate total of List elements
21       public static double sum(List<Number> list) {
22          double total = 0; // initialize total
23
24          // calculate sum
25          for (Number element : list) {
26             total += element.doubleValue();
27          }
28
29          return total;
30       }
31    }
```

```
numberList contains: [1, 2.4, 3, 4.1]
Total of the elements in numberList: 10.5
```

Fig. 13.7 | Totaling the numbers in a `List<Number>`.

Line 9 creates an array of `Number`s. The initializers are primitive values, so each is boxed as an object of its corresponding wrapper type—the `int` values `1` and `3` are boxed as `Integer` objects, and the `double` values `2.4` and `4.1` are boxed as `Double` objects. Line 10 creates a `List<Number>` from the array. Line 13 outputs the `List`'s `String` representation. Lines 16–17 display the total of the elements returned by the call `sum(numberList)`.

Method `sum` (lines 21–30) receives a `List<Number>` and totals its elements. The method uses `double` values to perform the calculations and returns the result as a `double`. Lines 25–27 total the `List`'s elements. The `for` statement

- assigns each `Number` to `element`,
- uses **`Number` method `doubleValue`** to obtain the `Number`'s underlying primitive value as a `double`, then
- adds the result to `total`.

When the loop terminates, line 29 returns the `total`.

A `List<Integer>` Is Not a `List<Number>`

You might think that because the `sum` method receives a `List<Number>` you could pass it a `List<Integer>` or a `List<Double>`

because `Integer` and `Double` inherit from `Number`. To test this, we modified Fig. 13.7's `TotalNumbers` class to create a `List<Integer>` containing the values 1–4 and passed it to method `sum`. The file `TotalNumbersErrors.java` in the `fig13_07` example folder contains the modified code.

When we compiled `TotalNumbersErrors.java` with the additional command-line option `-Xdiags:verbose` before the file name, the compiler issued the following more error message—the command-line option adds more detail:

Click here to view code image

```
TotalNumbersErrors.java:17: error: method sum in class
TotalNumbersErrors cannot be applied to given types;
         sum(numberList));
         ^
  required: List<Number>
  found:    List<Integer>
  reason: argument mismatch; List<Integer> cannot be converted to
          List<Number>
1 error
error: compilation failed
```

Though `Number` is `Integer`'s superclass, the compiler doesn't consider `List<Number>` to be a supertype of `List<Integer>`. But why? If `List<Integer>` had the *is-a* relationship with `List<Number>`, every operation we can perform on a `List<Number>` should also work on a `List<Integer>`. However, consider this: Adding a `Double` to a `List<Number>` works because **a `Double` *is a* `Number`**, and adding an `Integer` to a `List<Number>` works because **an `Integer` *is a* `Number`**, but adding a `Double` to a `List<Integer>` results in a compilation error because **a `Double` is not an `Integer`. The *is-a* relationship does not hold.**

13.7.2 Reimplementing Method `sum` with a Wildcard Type Argument in Its Parameter

How do we declare a more flexible `sum` method that can total the elements of any `List` containing elements of any subclass of `Number`? This is where **wildcard type arguments** are important. In Fig. 13.8, method `sum`'s parameter is declared in line 34 with the type:

```
List<? extends Number>
```

A wildcard type argument is denoted by a **question mark**, `?`, representing an "unknown type." In this case, the wildcard `extends` class `Number`, which means that `Number` is the wildcard's upper bound—the unknown-type argument must be either `Number` or one of its subclasses. With the wildcard type argument, method `sum` can receive a `List` containing any type of `Number`. To confirm that this works, we call `sum` with a `List<Integer>` (line 14), a `List<Double>` (line 22) and a `List<Number>` (line 30).

Click here to view code image

```
1  // Fig. 13.8: WildcardTest.java
2  // Wildcard test program.
3  import java.util.Arrays;
4  import java.util.List;
5
6  public class WildcardTest {
7     public static void main(String[] args) {
8        // create and initialize a List<Integer>
9        Integer[] integers = {1, 2, 3, 4, 5};
10       List<Integer> integerList = Arrays.asList(integers);
11
12       System.out.printf("integerList contains: %s%n", integerList);
13       System.out.printf("Total of the elements in integerList: %.0f%n%n",
14          sum(integerList));
15
16       // create and initialize a List<Double>
17       Double[] doubles = {1.1, 3.3, 5.5};
18       List<Double> doubleList = Arrays.asList(doubles);
19
20       System.out.printf("doubleList contains: %s%n", doubleList);
21       System.out.printf("Total of the elements in doubleList: %.1f%n%n",
22          sum(doubleList));
23
24       // create and initialize a List<Number> with mixed numeric types
25       Number[] numbers = {1, 2.4, 3, 4.1};
26       List<Number> numberList = Arrays.asList(numbers);
27
28       System.out.printf("numberList contains: %s%n", numberList);
29       System.out.printf("Total of the elements in numberList: %.1f%n",
30          sum(numberList));
31    }
32
33    // total the elements; using a wildcard in the List parameter
34    public static double sum(List<? extends Number> list) {
35       double total = 0; // initialize total
36
37       // calculate sum
38       for (Number element : list) {
```

```
            total += element.doubleValue();
        }

        return total;
    }
}
```

```
integerList contains: [1, 2, 3, 4, 5]
Total of the elements in integerList: 15

doubleList contains: [1.1, 3.3, 5.5]
Total of the elements in doubleList: 9.9

numberList contains: [1, 2.4, 3, 4.1]
Total of the elements in numberList: 10.5
```

Fig. 13.8 | Wildcard test program.

Lines 9–14 create and initialize a `List<Integer>`, output its elements and total them by calling method `sum` (line 14). Lines 17–22 and 25–30 perform the same operations for a `List<Double>` then a `List<Number>` containing both `Integer`s and `Double`s. In method `sum` (lines 34–43), although the `List` argument's element types are not directly known by the method, they're known to be at least of type `Number`—the wildcard's upper bound. So, line 39 compiles because all `Number` objects are guaranteed to have the `doubleValue` method.

Wildcard Restrictions

The wildcard (`?`) in the method's header (line 34) does not specify a type-parameter name, so you cannot use it as a type name throughout the method's body (i.e., you cannot replace `Number` with `?` in line 38). You could, however, declare `sum` as follows:

Click here to view code image

```
public static <T extends Number> double sum(List<T> list)
```

which allows the method to receive a `List` that contains elements of any `Number` subclass—the code with this update is in `WildcardTest2.java` in the `fig13_08` folder. You could then use the type parameter `T` throughout the method body and in the return type.

If the wildcard is specified without an upper bound, then only the methods of type `Object` can be invoked on values of the

wildcard type. Also, methods that use wildcards in their parameters' type arguments cannot add elements to collections referenced by the parameters. Using a wildcard in a method's type-parameter section or using a wildcard as an explicit type of a variable in the method body is a syntax error.

Lower Bound of a Wildcard

Just as you can specify a wildcard type argument's upper bound with `extends`, you can specify a wildcard type argument's **lower bound** with `super`. For example, a method parameter declared as

```
List<? super Integer>
```

indicates that the corresponding argument can be a `List<Integer>` or a `List` containing one of `Integer`'s supertypes, such as `List<Number>` or `List<Object>`. You'll explore this further in Generative AI question 5.

Generative AI

1 Prompt genAIs, asking why Java generic methods that use wildcards with upper bounds in their parameter's type arguments cannot add elements to a collection referenced by the parameter.
2 Prompt genAIs, asking for suggestions on when to use Java wildcards in generic methods and classes.
3 Prompt genAIs, asking for insights on how wildcards in Java generics affect type safety.
4 Prompt genAIs, asking for simple examples of Java API generic methods or classes having wildcard type arguments with an upper bound.
5 Prompt genAIs, asking for simple examples of Java API generic methods or classes having wildcard type arguments with a lower bound.
6 Prompt genAIs, asking when you would choose a Java wildcard type argument with an upper bound vs. a lower bound.

13.8 Wrap-Up

This chapter introduced custom generic methods and classes. First, we motivated generic methods by replacing overloaded methods—such as those for printing different arrays—with a single generic

method. We declared a generic method with a type parameter section (`<T>`) that precedes the method's return type then used the type parameter (`T`) throughout the method declaration. You learned that generic methods improve reusability and maintainability by applying the same logic to multiple types.

We used type bounds in generic methods to limit the types passed as arguments. For example, our `maximum` method determined the largest of three values of a type that implements the `Comparable<T>` interface, preventing errors that would occur if non-comparable types were passed.

We discussed how compile-time type erasure replaces type parameters with actual types. In particular, if you do not restrict the type parameter with an upper bound, the compiler replaces it with `Object` by default; otherwise, it uses the upper bound. You learned that generic methods can be overloaded by both generic and non-generic methods, and that the compiler uses the most specific match available for each method call.

We presented a generic `Stack` class that could store objects of any type. We instantiated the `Stack<E>` for specific types like `Stack<Integer>` and `Stack<Double>`. You learned that the generic `Stack` provides type safety by preventing incompatible types from being pushed onto the stack. We refactored the non-generic methods in the `StackTest` program into generic methods to reduce redundancy.

We used wildcards to increase a generic method's flexibility. We specified an upper bound with a question mark (`?`) followed by `extends`. We created a `sum` method that could receive a `List` containing elements of any type that has the *is-a* relationship with `Number`, enabling method `sum` to receive `List`s for each subclass of `Number`. The next several chapters present JavaFX and how it's used to build applications with event-driven graphical user interfaces (GUIs), graphics and multimedia.

Special Section: Generative AI Exercises

For each of the following questions, try multiple genAIs to determine which performs best. Always verify the results. Keep in mind that they sometimes make mistakes and even hallucinate, creating unreliable information.

13.1 ***(Exceptions and Generics)*** Prompt genAIs, asking them to discuss best practices for exception handling when working with

Java generics.

13.2 ***(Overloaded Generic Method printArray)*** Prompt genAIs, asking them to overload Fig. 13.2's generic method `printArray` to take the additional `int` arguments `lowIndex` and `highIndex`. This method should print only the designated range of elements from `lowIndex` up to, but not including, `highIndex`. Validate `lowIndex` and `highIndex`. If either is out of range, throw an `InvalidSubscriptException`. Modify Fig. 13.2's `main` method to use both `printArray` versions with `integerArray` and `doubleArray`.

13.3 ***(Method maximum with a Variable-Length Argument List)*** Prompt genAIs, asking them to modify Fig. 13.3's method `maximum` to receive one or more arguments. Calling the method with no arguments should produce a compilation error. This method should have two parameters with the second representing a variable-length argument list. Test the generated method.

13.4 ***(Generic maximumElement and minimumElement Methods)*** Prompt genAIs, asking them to create generic methods `maximumElement` and `minimumElement` that return the largest and smallest value in a `Collection`. Each should be able to receive `Collection`s of any class that implements interface `Comparable<T>`. Test the generated method with `List` and `Set` collections containing comparable elements.

Functional Programming

with Lambdas & Streams

Objectives

In this chapter, you'll:

- Learn various functional-programming techniques and how they complement object-oriented programming.
- Use lambdas and streams to simplify tasks that process sequences of elements.
- Learn what streams are and how stream pipelines are formed from stream sources, intermediate operations and terminal operations.
- Create streams representing ranges of `int` values and random `int` values.
- Implement functional interfaces with lambdas.
- Perform on `IntStream`s intermediate operations `filter`, `map`, `mapToObj` and `sorted`, and terminal operations `forEach`, `count`, `min`, `max`, `sum`, `average` and `reduce`.
- Perform on `Stream`s intermediate operations `distinct`, `filter`, `map`, `mapToDouble` and `sorted`, and terminal operations `collect`, `forEach`, `findFirst` and `reduce`.
- Process infinite streams.

Outline

14.1 Introduction

14.2 Streams and Reduction

14.2.1 Summing the Integers from 1 through 10 with a `for` Loop

14.2.2 External Iteration with `for` Is Error Prone

14.2.3 Summing with a Stream and Reduction

14.2.4 Internal Iteration

14.3 Mapping and Lambdas

14.3.1 Lambda Expressions

14.3.2 Lambda Syntax
14.3.3 Intermediate and Terminal Operations
14.4 Filtering
14.5 How Elements Move Through Stream Pipelines
14.6 Method References
14.6.1 Creating an `IntStream` of Random Values
14.6.2 Performing a Task on Each Stream Element with `forEach` and a Method Reference
14.6.3 Mapping Integers to String Objects with `mapToObj`
14.6.4 Concatenating Strings with `collect`
14.7 `IntStream` Operations
14.7.1 Creating an `IntStream` and Displaying Its Values
14.7.2 Terminal Operations `count`, `min`, `max`, `sum` and `average`
14.7.3 Terminal Operation `reduce`
14.7.4 Sorting `IntStream` Values
14.8 Functional Interfaces
14.9 Lambdas: A Deeper Look
14.10 `Stream<Integer>` Manipulations
14.10.1 Creating a `Stream<Integer>`, Sorting It and Collecting the Results
14.10.2 Filtering a `Stream` and Storing the Results for Later Use
14.10.3 Filtering and Sorting a `Stream` and Collecting the Results
14.10.4 Sorting Previously Collected Results
14.11 `Stream<String>` Manipulations
14.11.1 Mapping `String`s to Uppercase
14.11.2 Filtering `String`s Then Sorting Them in Case-Insensitive Ascending Order
14.11.3 Filtering `String`s Then Sorting Them in Case-Insensitive Descending Order
14.12 `Stream<Employee>` Manipulations
14.12.1 Creating and Displaying a `List<Employee>`
14.12.2 Filtering `Employee`s with Salaries in a Specified Range

14.12.3 Sorting `Employee`s By Multiple Fields
14.12.4 Mapping `Employee`s to Unique-Last-Name `String`s
14.12.5 Grouping `Employee`s By Department
14.12.6 Counting the Number of `Employee`s in Each Department
14.12.7 `Summing` and Averaging `Employee` Salaries
14.13 Creating a `Stream<String>` from a File
14.14 Streams of Random Values
14.15 Infinite Streams
14.16 Additional Notes on Interfaces
14.17 Wrap-Up
Special Section: Generative AI Exercises

14.1 Introduction[1]

1. We were privileged to have Brian Goetz, Oracle's Java Language Architect and Specification Lead for Java's Project Lambda, and co-author of *Java Concurrency in Practice*, do a detailed review of *Java How to Program, 10/e*. He thoroughly scrutinized the version of this chapter that appeared in that edition and provided many additional suggestions that are reflected in this new edition of *Java for Programmers*. Any remaining faults in the book are our own.

So far we've discussed three programming paradigms—procedural, object-oriented and generic. In this chapter, we present lambdas and streams,[2] the technologies used with Java's fourth paradigm—functional programming.

2. The streams we discuss in this chapter are not the same as the input/output streams we discuss in Chapter 11, Files, I/O Streams, JSON Serialization & CSV Files, in which a program reads a stream of bytes from or writes a stream of bytes to a file. Section 14.13 uses lambdas and streams to manipulate the contents of a file.

We'll use lambdas and streams to write certain kinds of programs faster, simpler, more concisely and with fewer bugs than previous techniques. In Chapter 18, Concurrency: Platform Threads to Virtual Threads, you'll learn that such programs can be easier to parallelize (i.e., perform multiple operations simultaneously) so that you can take advantage of multi-core architectures to enhance performance—a key goal of lambdas and streams.

In Chapter 18, you'll also learn that it's hard to create parallel tasks that operate correctly if those tasks modify a program's state

(that is, its variables' values). So, the techniques you'll learn in this chapter focus on **immutability**—not modifying the data source being processed or any other program state.

This chapter presents many examples of lambdas and streams (summarized in the following table), beginning with several demonstrating better ways to implement tasks you programmed in Chapter 4. The first several examples are presented in a manner that allows them to be used in the context of earlier chapters. For this reason, some terminology is discussed later in this chapter.

Section	Covered after
Sections 14.2–14.4	Chapters 4, 5 and 6
Section 14.5	Chapter 5
Section 14.6	Chapter 6
Sections 14.8–14.9	Chapters 9, 10 and 12
Section 14.10	Chapter 9
Section 14.11	Chapter 14

- Sections 14.2–14.4 introduce basic capabilities that process integer ranges and eliminate the need for counter-controlled iteration.
- Section 14.5 introduces method lambdas and streams to process integer ranges
- Section 14.6 presents lambda and process one-dimensional arrays.
- Sections 14.8–14.9 discuss functional lambda concepts, tying these examples. (Section 9.12 introduced the functional interfaces concept.)
- Section 14.10 uses lambdas and `String` objects.
- Section 14.11 uses lambdas and from a file—the example in this section also uses regular expression capabilities from Chapter 7.

Chapter 18 uses parallel stream processing to demonstrate that functional programs are easier to parallelize to enhance performance on multi-core architectures. We'll also demonstrate `Arrays` methods `parallelSort` and `sort`, showing that `parallelSort` can improve sorting performance on today's multi-core systems.

14.2 Streams and Reduction

[This section demonstrates how streams can simplify programming tasks you learned in Chapter 4, Control Statements: Part 2.]

In counter-controlled iteration, you typically determine *what* you

want to accomplish then specify precisely *how* to accomplish it using a `for` loop. In this section, we'll investigate that approach then present a better way to accomplish the same tasks.

14.2.1 Summing the Integers from 1 through 10 with a `for` Loop

Let's assume that *what* you want to accomplish is to sum the integers from 1 through 10. You can do this using a counter-controlled `for` loop (introduced in Chapter 4):

Click here to view code image

```
int total = 0;

for (int number = 1; number <= 10; ++number) {
   total += number;
}
```

This loop specifies precisely *how* to perform the task—with a `for` statement that processes each value of `number` from 1 through 10, adding its current value to `total` once per loop iteration and incrementing `number` after each addition operation. This is known as **external iteration** because *you* specify all the iteration details.

14.2.2 External Iteration with `for` Is Error Prone

Let's consider potential problems with the preceding code. As implemented, the loop requires two variables (`total` and `number`) that the code mutates (modifies) during each loop iteration. Every time you write code that modifies a variable, it's possible to introduce an error into your code. There are several opportunities for error in the preceding code. For example, you could:

- initialize the variable `total` incorrectly,
- initialize the `for` loop's control variable `number` incorrectly,
- use the wrong loop-continuation condition,
- increment control variable `number` incorrectly or
- incorrectly add each value of `number` to the `total`.

In addition, as the tasks you perform get more complicated,

understanding *how* the code works gets in the way of understanding *what* it does. This makes the code harder to read, debug and modify, and more likely to contain errors.

14.2.3 Summing with a Stream and Reduction

Now, let's take a different approach, specifying *what* to do rather than *how* to do it. In Fig. 14.1, we specify *what* we want to accomplish—sum the integers from 1 through 10—then let Java's **`IntStream`** class (package **`java.util.stream`**) determine *how* to do it. The key to this program is the following expression in lines 8–9

```
IntStream.rangeClosed(1, 10)
         .sum()
```

Click here to view code image

```
1  // Fig. 14.1: StreamReduce.java
2  // Sum the integers from 1 through 10 with IntStream.
3  import java.util.stream.IntStream;
4
5  public class StreamReduce {
6     public static void main(String[] args) {
7        System.out.printf("Sum of 1 through 10 is: %d%n",
8           IntStream.rangeClosed(1, 10)
9                    .sum());
10    }
11 }
```

```
Sum of 1 through 10 is: 55
```

Fig. 14.1 | Sum the integers from 1 through 10 with `IntStream`.

which can be read as, "for the stream of `int` values in the range 1 through 10, calculate the sum" or "sum the numbers from 1 through 10." In this code, notice that there is neither a counter-control variable nor a variable to store the total—`IntStream` encapsulates those details in `rangeClosed` and `sum`.

Streams and Stream Pipelines

The chained method calls in lines 8–9 create a **stream pipeline**. A **stream** is a sequence of elements on which you perform tasks. The stream pipeline moves the stream's elements through a sequence of

tasks or processing steps. When using chained method calls, align the dots (`.`) vertically for readability.

Specifying the Data Source

A stream pipeline typically begins with a method call that creates the stream—this is known as the data source. Line 8 specifies the data source with the method call

```
IntStream.rangeClosed(1, 10)
```

which creates an `IntStream` representing an ordered range of `int` values.

The **`IntStream static` method `rangeClosed`** creates an `IntStream` representing the ordered sequence of `int` elements `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8`, `9` and `10`. The method is named `rangeClosed` because it produces a closed range of values that includes both arguments (`1` and `10`). `IntStream` also provides **method `range`**, which produces a half-open range of values from its first argument up to, but not including, its second argument—for example,

```
IntStream.range(1, 10)
```

produces an `IntStream` representing the ordered sequence of `int` elements `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8` and `9`, but not `10`.

Calculating the Sum of the `IntStream`'s Elements

Next, line 9 completes the stream pipeline with the processing step

```
.sum()
```

This invokes **`IntStream`'s `sum` instance method**, which returns the sum of all the `int`s in the stream—in this case, the sum of the integers from 1 through 10.

The processing step performed by method `sum` is a **reduction**—it reduces the stream of values to a single value (the sum). This is one of several predefined `IntStream` reductions. Section 14.7 presents the predefined reduction methods `count`, `min`, `max`, `average` and `summaryStatistics`, and the `reduce` method for defining custom reductions.

Processing the Stream Pipeline

A **terminal operation** initiates a stream pipeline's processing and

produces a result. The `IntStream` method `sum` is a terminal operation that returns the sum of the stream's elements. Similarly, `count`, `min`, `max`, `average`, `summaryStatistics` and `reduce` are all terminal operations. We'll present others throughout this chapter. Section 14.3.3 discusses terminal operations in more detail.

14.2.4 Internal Iteration

The key to the preceding example is that it specifies *what* we want the task to accomplish—calculating the sum of the integers from 1 through 10—rather than *how* to accomplish it. This is an example of **declarative programming** (specifying *what*) vs. **imperative programming** (specifying *how*).

We broke the goal into two simple tasks:

- producing the numbers in a closed range (1–10) and
- calculating their sum.

Internally, `IntStream` already knows how to perform each. We did not need to specify *how* to iterate through the elements or declare and use any mutable variables. This is known as **internal iteration** because `IntStream` handles all the iteration details—an essential aspect of **functional programming**. Unlike external iteration with the `for` statement, the primary potential for error in line 8 of Fig. 14.1 is specifying the incorrect starting and/or ending values as the `rangeClosed` method's arguments. Once you're used to it, a stream pipeline's code can also be easier to read.

Functional-programming techniques enable you to write higher-level code because many of the details are implemented for you by the Java streams library. Your code becomes more concise, which improves productivity and can help you rapidly prototype programs. These techniques also eliminate large classes of errors, such as

- off-by-one errors—because iteration details are hidden from you and handled by the libraries, and
- incorrectly modifying variables—because you focus on immutability and thus do not modify data.

This makes it easier to write correct programs.

Generative AI

1 Prompt genAIs to reimplement the stream pipeline in Fig. 14.1 using `IntStream` method `range` rather than `rangeClosed`.
2 Prompt genAIs to use code and sample outputs to explain common reductions Java programmers create with lambdas and streams.
3 Prompt genAIs to discuss common errors you can avoid by using Java streams and functional-programming techniques.
4 Prompt genAIs to discuss the performance costs of immutability when using Java streams and functional-programming techniques.

14.3 Mapping and Lambdas

[This section demonstrates how streams simplify programming tasks you learned in Chapter 4, Control Statements: Part 2.]

The preceding example specified a stream pipeline containing only a data source and a terminal operation that produced a result. Most stream pipelines also contain **intermediate operations** specifying tasks to perform on a stream's elements before a terminal operation produces a result.

Next, we introduce the **mapping intermediate operation** to transform a stream's elements to new values. The result is a stream with the same number of elements containing the transformation's results. Sometimes, the mapped elements are of different types from the original stream's elements.

To demonstrate mapping, let's revisit the program of Fig. 4.3 in which we calculated the sum of the even integers from 2 through 20 using external iteration, as follows:

Click here to view code image

```
int total = 0;

for (int number = 2; number <= 20; number += 2) {
   total += number;
}
```

Figure 14.2 reimplements this task using streams and internal iteration.

Click here to view code image

```
1  // Fig. 14.2: StreamMapReduce.java
2  // Sum the even integers from 2 through 20 with IntStream.
3  import java.util.stream.IntStream;
4
5  public class StreamMapReduce {
6     public static void main(String[] args) {
7        System.out.printf("Sum of the even ints from 2 through 20 is: %d%n",
8           IntStream.rangeClosed(1, 10)              // 1...10
9                    .map((int x) -> {return x * 2;}) // multiply by 2
10                   .sum());                         // sum
11    }
12 }
```

```
Sum of the even ints from 2 through 20 is: 110
```

Fig. 14.2 | Sum the even integers from 2 through 20 with `IntStream`.

The stream pipeline in lines 8–10 performs three chained method calls:

- Line 8 creates the data source—an `IntStream` containing the elements `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8`, `9` and `10`.
- Line 9, which we'll discuss in detail momentarily, performs a processing step that maps each element (`x`) in the stream to that element multiplied by 2. The result is a stream of the even integers `2`, `4`, `6`, `8`, `10`, `12`, `14`, `16`, `18` and `20`.
- Line 10 reduces the stream's elements to a single value—the `sum` of the elements. This terminal operation initiates the pipeline's processing and `sum`s the stream's elements.

The new feature here is line 9's mapping operation, which, in this example, multiplies each stream element by 2. **`IntStream` method `map`** receives as its argument (line 9)

```
(int x) -> {return x * 2;}
```

This notation, discussed in Section 14.3.1, represents "a method that receives an `int` parameter `x` and returns that value multiplied by `2`." For each element in the stream, `map` calls this method, passing the current stream element to it. The method's return value becomes part of the new stream that `map` returns.

14.3.1 Lambda Expressions

Throughout this chapter, we'll demonstrate many intermediate and terminal stream operations that receive methods as arguments. Method `map`'s argument in line 9

```
(int x) -> {return x * 2;}
```

is a lambda expression (or simply "a lambda"), which represents an **anonymous method**—a method without a name. Though a lambda expression's syntax does not look like the methods we've presented previously, the left side looks like a method parameter list, and the right side looks like a method body. We explain the syntax details shortly.

Lambda expressions enable you to create methods that can be treated as data. You can:

- pass lambdas as arguments to other methods (like `map` or even other lambdas),
- assign lambda expressions to variables for later use, and
- return lambda expressions from methods.

We'll demonstrate these powerful capabilities. Lambdas and streams enable you to combine many benefits of functional-programming techniques with the benefits of object-oriented programming.

14.3.2 Lambda Syntax

A lambda consists of a parameter list followed by the **arrow token (->)** and a body, as in:

```
(parameters) -> {statements}
```

The lambda in line 9

```
(int x) -> {return x * 2;}
```

receives an `int`, multiplies its value by `2` and returns the result. In this case, the body is a statement block that may contain statements enclosed in curly braces. The compiler infers from the lambda that it returns an `int`, because the parameter `x` is an `int` and the literal `2` is an `int`—multiplying an `int` by an `int` yields an `int` result. As in a method declaration, lambdas specify parameters in a comma-separated list. The preceding lambda is similar to the method

```
int multiplyBy2(int x) {
   return x * 2;
}
```

but the lambda does not have a name, and the compiler infers its return type. There are several variations of the lambda syntax.

Eliminating a Lambda's Parameter Type(s)

A lambda's parameter type(s) usually may be omitted, as in:

```
(x) -> {return x * 2;}
```

in which case, the compiler infers the parameter and return types by the lambda's context—we'll say more about this later. If the compiler cannot infer the parameter or return types (e.g., if there are multiple type possibilities), it generates an error.

Simplifying the Lambda's Body

If the body contains only one expression, the `return` keyword, curly braces and semicolon may be omitted, as in:

```
(x) -> x * 2
```

In this case, the lambda implicitly returns the expression's value.

Simplifying the Lambda's Parameter List

If the parameter list contains only one parameter, the parentheses may be omitted, as in:

```
x -> x * 2
```

Lambdas with Empty Parameter Lists

To define a lambda with an empty parameter list, use empty parentheses to the left of the arrow token (`->`), as in:

Click here to view code image

```
() -> System.out.println("Welcome to lambdas!")
```

Method References

In addition to the preceding lambda-syntax variations, there are specialized shorthand forms of lambdas known as method

references, which we introduce in Section 14.6.

14.3.3 Intermediate and Terminal Operations

In the stream pipeline of lines 8–10 (Fig. 14.2), `map` is an intermediate operation, and `sum` is a terminal operation. Method `map` is one of many intermediate operations that specify tasks to perform on a stream's elements.

Lazy and Eager Operations

Intermediate operations use **lazy evaluation**—each intermediate operation results in a new stream object, but no operations are performed on the stream's elements until a terminal operation is called to produce a result. This allows library developers to optimize stream-processing performance. For example, if you have 1,000,000 `Person` objects and you're looking for the first one with the last name `"Jones"`, rather than processing all 1,000,000 elements, stream processing can terminate as soon as the first matching `Person` object is found. Lazy evaluation helps improve performance by ensuring that operations are performed only if necessary.

Terminal operations are **eager**—they perform the requested operation when called. We say more about lazy and eager operations as we encounter them throughout the chapter. We'll discuss how lazy operations can improve performance in Section 14.5, which demonstrates how a stream pipeline's intermediate operations are applied to each stream element. The following table contains some common intermediate operations.

Common intermediate stream operations

`filter`	Returns a stream containing only the elements that satisfy a condition (known as a predicate). The new stream often has fewer elements than the original stream.
`distinct`	Returns a stream containing only the unique elements—duplicates are eliminated.
`limit`	Returns a stream with the specified number of elements from the beginning of the original stream.
`map`	Returns a stream in which each of the original stream's elements is mapped to a new value (possibly of a different type)—for example, mapping numeric values to the squares of the numeric values or mapping numeric grades to letter grades (A, B C, D or F). The new stream has the same number of elements as the

original stream.

Returns a stream in which the elements are in sorted order. The new stream has the same number of elements as the original stream. We'll show how to specify both ascending and descending order.

The following table contains some common terminal operations.[3]

3. For a complete list of the intermediate and terminal operations, visit `https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/util/stream/Stream.html`. Each `Stream` method's documentation indicates whether it's an intermediate or terminal operation. Java 23 includes as a preview feature stream gatherers, which enable you to create custom intermediate operations. This feature is expected to be finalized in Java 24. For more information, visit `https://openjdk.org/jeps/473`.

Common terminal stream operations

Performs processing on every element in a stream (for example, display each element).

Reduction operations—Take all values in the stream and return a single value

Returns the average of the elements in a numeric stream.

Returns the number of elements in the stream.

Returns the maximum value in a stream.

Returns the minimum value in a stream.

Reduces the elements of a collection to a single value using an associative accumulation function—that is, a function that produces the same result regardless of the order in which operations are applied to its arguments. For example, in a lambda that adds two elements and returns the sum, the addition operation is associative—`x + y` and `y + x` produce the same result.

14.4 Filtering

[This section demonstrates how streams can be used to simplify programming tasks that you learned in Chapter 4, Control Statements: Part 2.]

Another common intermediate stream operation is filtering

elements to select those that match a condition—known as a predicate. For example, the following code selects the even integers in the range 1–10, doubles them and sums the results:

Click here to view code image

```
int total = 0;

for (int x = 1; x <= 10; ++x) {
   if (x % 2 == 0) { // if x is even
      total += x * 2;
   }
}
```

Figure 14.3 reimplements this loop using streams.

Click here to view code image

```
1  // Fig. 14.3: StreamFilterMapReduce.java
2  // Double the even ints from 2 through 10 then sum them with IntStream.
3  import java.util.stream.IntStream;
4
5  public class StreamFilterMapReduce {
6     public static void main(String[] args) {
7        System.out.printf(
8           "Sum of the doubles of the even ints from 2 through 10 is: %d%n",
9           IntStream.rangeClosed(1, 10)
10                   .filter(x -> x % 2 == 0)
11                   .map(x -> x * 2)
12                   .sum());
13    }
14 }
```

```
Sum of the doubles of the even ints from 2 through 10 is: 60
```

Fig. 14.3 | Double the even `int`s from 2 through 10 then sum them with `IntStream`.

The stream pipeline in lines 9–12 performs four chained method calls:

- Line 9 creates the data source—an `IntStream` for the closed range 1 through 10.
- Line 10, which we'll discuss in detail momentarily, filters the stream's elements by selecting only the elements that are divisible by 2 (that is, the even integers), producing a stream of the even integers from 2, 4, 6, 8 and 10.

- Line 11 maps each element (`x`) in the stream to that element times 2, producing a stream of the even integers 4, 8, 12, 16 and 20.
- Line 12 reduces the stream to the sum of its elements (60).

The new feature here is the filtering operation in line 10. **`IntStream` method `filter`** receives as its argument a method that takes one parameter and returns a `boolean` result. If the result is `true` for a given element, that element is included in the resulting stream.

The lambda in line 10:

```
x -> x % 2 == 0
```

determines whether its `int` argument is divisible by `2` (that is, the remainder after dividing by `2` is `0`) and, if so, returns `true`; otherwise, the lambda returns `false`. For each element in the stream, `filter` calls the method it receives as an argument, passing to the method the current stream element. If the method's return value is `true`, the corresponding element becomes part of the intermediate stream that `filter` returns.

Line 10 creates an intermediate stream representing only the elements that are divisible by `2`. Next, line 11 uses `map` to create an intermediate stream representing the even integers (2, 4, 6, 8 and 10) multiplied by 2 (4, 8, 12, 16 and 20). Line 12 initiates the stream processing with a call to the terminal operation `sum`. At this point, the combined processing steps are applied to each element, then `sum` returns the total of the remaining elements in the stream. We discuss this further in the next section.

Order of Pipeline Operations is Important

The order of the operations in a stream pipeline matters:

- `filter`ing the even numbers from 1–10 yields 2, 4, 6, 8 and 10, then `map`ping them to twice their values yields 4, 8, 12, 16 and 20.
- On the other hand, `map`ping the numbers from 1–10 to twice their values yields 2, 4, 6, 8, 10, 12, 14, 16, 18 and 20, then `filter`ing the even numbers gives all of those values because they're all even before the `filter` operation is performed.

The stream pipeline in this example could have been implemented using only `map` and `sum`.

Generative AI

1 Prompt genAIs with Fig. 14.3's code asking them to modify the program to calculate the sum of the squares of the odd integers.
2 Prompt genAIs to explain the performance advantages of lazy evaluation in the context of Fig. 14.3's code.
3 Prompt genAIs to reimplement Fig. 14.3 using only `map` and `sum`.

14.5 How Elements Move Through Stream Pipelines

Section 14.3 mentioned that each intermediate operation results in a new stream. Each new stream is simply an object representing the processing steps specified to that point in the pipeline. Chaining method calls for intermediate operations adds to the set of processing steps to perform on each stream element. The last stream object in the stream pipeline contains all the processing steps to perform on each stream element.

When you initiate a stream pipeline with a terminal operation, the intermediate operations' processing steps are applied for a given stream element before they are applied to the next stream element. So, the stream pipeline in Fig. 14.3 operates as follows:

Click here to view code image

```
For each element
    If the element is an even integer
        Multiply the element by 2 and add the result to the total
```

To prove this, consider a modified version of Fig. 14.3's stream pipeline in which each lambda displays the intermediate operation's name and the current stream element's value:

Click here to view code image

```
IntStream.rangeClosed(1, 10)
         .filter(
            x -> {
               System.out.printf("%nfilter: %d%n", x);
               return x % 2 == 0;
            })
         .map(
            x -> {
               System.out.println("map: " + x);
               return x * 2;
```

```
        })
    .sum()
```

It's clear from the modified pipeline's output below (we added the comments) that each even integer's `map` step is applied before the next stream element's `filter` step:

Click here to view code image

```
filter: 1 // odd so no map step is performed for this element

filter: 2 // even so a map step is performed next
map: 2

filter: 3 // odd so no map step is performed for this element

filter: 4 // even so a map step is performed next
map: 4

filter: 5 // odd so no map step is performed for this element

filter: 6 // even so a map step is performed next
map: 6

filter: 7 // odd so no map step is performed for this element

filter: 8 // even so a map step is performed next
map: 8

filter: 9 // odd so no map step is performed for this element

filter: 10 // even so a map step is performed next
map: 10
```

For the odd elements, the `map` step was not performed. When a `filter` step returns `false`, the element's remaining processing steps are ignored because that element is not included in the results. (This version of Fig. 14.3 is located in a subfolder with that example.)

Generative AI

1 Prompt genAIs with the code from the modified version of Fig. 14.3, asking them to replace the `sum` call with a custom reduction that sums the values produced by the `map` step and prints `"reduce: "` followed by the value the reduction step adds to the total.

14.6 Method References

[This section demonstrates how streams can be used to simplify programming tasks that you learned in Chapter 5, Methods.]

For a lambda that simply calls another method, you can replace the lambda with that method's name—known as a **method reference**. The compiler converts a method reference into an appropriate lambda expression.

Like Fig. 5.2, Fig. 14.4 uses `RandomGenerator` to obtain random numbers in the range 1–6. The program uses streams to create the random values and method references to help display the results. We discuss the code in Sections 14.6.1–14.6.4.

Click here to view code image

```
1  // Fig. 14.4: RandomIntegers.java
2  // Shifted and scaled random integers.
3  import java.util.random.RandomGenerator;
4  import java.util.stream.Collectors;
5
6  public class RandomIntegers {
7     public static void main(String[] args) {
8        var random = RandomGenerator.getDefault();
9
10       // display 10 random integers on separate lines
11       System.out.println("Random numbers on separate lines:");
12       random.ints(10, 1, 7)
13             .forEach(System.out::println);
14
15       // display 10 random integers on the same line
16       String numbers =
17          random.ints(10, 1, 7)
18                .mapToObj(String::valueOf)
19                .collect(Collectors.joining(" "));
20       System.out.printf("%nRandom numbers on one line: %s%n", numbers);
21
22    }
23 }
```

```
Random numbers on separate lines:
4
3
4
5
1
```

```
5
5
3
6
5

Random numbers on one line: 4 6 2 5 6 4 3 2 4 1
```

Fig. 14.4 | Shifted and scaled random integers.

14.6.1 Creating an `IntStream` of Random Values

Class **`RandomGenerator`'s `ints` method** returns an `IntStream` of random numbers. In the stream pipeline of line 12

```
random.ints(10, 1, 7)
```

creates an `IntStream` data source with the specified number of random `int` values (`10`) in the range starting with the first argument (`1`) up to, but not including, the second argument (`7`). So, line 12 produces an `IntStream` of `10` random integers in the range `1`–`6`.

14.6.2 Performing a Task on Each Stream Element with `forEach` and a Method Reference

Next, line 13 of the stream pipeline uses **`IntStream` method `forEach`** (a terminal operation) to perform a task on each stream element. Method `forEach` receives as its argument a method that takes one parameter and performs a task using the parameter's value.

The `System.out::println` argument to `forEach` is a **method reference**—a shorthand notation for a lambda that calls the specified method. A method reference of the form

Click here to view code image

```
objectName::instanceMethodName
```

is a **bound instance method reference**—"bound" means the object (`System.out`) to the left of `::` calls the instance method (`println`) to

the right.

The compiler converts `System.out::println` into a one-parameter lambda like

```
x -> System.out.println(x)
```

that passes the lambda's argument—the current stream element (represented by `x`)—to the `System.out` object's `println` instance method, which outputs its argument's `String` representation. The stream pipeline of lines 12–13 is equivalent to the following `for` loop:

Click here to view code image

```
for (int i = 1; i <= 10; ++i) {
   System.out.println(random.nextInt(1, 7));
}
```

14.6.3 Mapping Integers to String Objects with `mapToObj`

The stream pipeline in lines 16–19

Click here to view code image

```
String numbers =
    random.ints(10, 1, 7)
          .mapToObj(String::valueOf)
          .collect(Collectors.joining(" "));
```

creates a `String` containing 10 random integers in the range 1–6 separated by spaces. The pipeline performs three chained method calls:

- Line 17 creates the data source—an `IntStream` of 10 random integers from 1–6.
- Line 18 maps each `int` to its `String` representation, resulting in an intermediate stream of `String`s. The `IntStream` method `map` we've used previously returns an `IntStream`. Here, we use the **`IntStream` method `mapToObj`**, which enables you to map from a stream of primitive-type elements to a stream of reference-type elements. Like `map`, `mapToObject` expects a one-parameter method that returns a result. In this example, `mapToObj`'s argument is a **`static` method reference** of the form *ClassName*`::`*staticMethodName*. `String static` method

`valueOf` returns its argument's `String` representation. The compiler converts the method reference `String::valueOf` into a one-parameter lambda that calls `valueOf`, passing the current stream element as an argument, as in

```
x -> String.valueOf(x)
```

- Line 19, which we discuss in more detail in Section 14.6.4, uses the **`Stream` terminal operation `collect`** to concatenate all the `String`s, separating each from the next with a space. Method `collect` is a form of reduction because it returns one object—in this case, a `String`.

Line 20 displays the resulting `String`.

14.6.4 Concatenating Strings with `collect`

Consider line 19 of Fig. 14.4. The `Stream` terminal operation `collect` uses a collector to gather the stream's elements into a single object—often a collection. This is similar to a reduction, but `collect` returns an object containing the stream's elements, whereas `reduce` returns a single value of the stream's element type. In this example, we use a predefined collector returned by the **`static` `Collectors` method `joining`**. This collector creates a concatenated `String` representation of the stream's elements, appending each element to the `String` separated from the previous element by the `joining` method's argument (in this case, a space). Method `collect` then returns the resulting `String`. We discuss other collectors throughout this chapter.

Generative AI

1 Prompt genAIs for brief descriptions of the most commonly used collectors defined by Java's `Collectors` class.
2 Prompt genAIs to explain the differences between method references and lambdas and when to prefer each.

14.7 `IntStream` Operations

[This section demonstrates how lambdas and streams can be used to simplify programming tasks like those you learned in Chapter 6, Arrays and `ArrayList`s.]

Figure 14.5 demonstrates additional `IntStream` operations on streams created from the array of `int`s named `values` in line 8. **`LongStream`**s and **`DoubleStream`**s (for `long` and `double` values, respectively) also support the `IntStream` techniques demonstrated so far. We split this example's code into parts discussed in Sections 14.7.1–14.7.4.

Click here to view code image

```
1  // Fig. 14.5: IntStreamOperations.java
2  // Demonstrating IntStream operations.
3  import java.util.stream.Collectors;
4  import java.util.stream.IntStream;
5
6  public class IntStreamOperations {
7     public static void main(String[] args) {
8        int[] values = {3, 10, 6, 1, 4, 8, 2, 5, 9, 7};
9
```

Fig. 14.5 | Demonstrating `IntStream` operations.

14.7.1 Creating an `IntStream` and Displaying Its Values

Lines 12–15 use a stream pipeline to display `int` array `values`' elements:

Click here to view code image

```
10        // display original values
11        System.out.print("Original values: ");
12        System.out.println(
13           IntStream.of(values)
14                    .mapToObj(String::valueOf)
15                    .collect(Collectors.joining(" ")));
16
```

```
Original values: 3 10 6 1 4 8 2 5 9 7
```

In the stream pipeline, **`IntStream static` method `of`** (line 13) receives an `int` array argument and returns an `IntStream` for processing the array's values. In lines 13–15:

- line 13 creates an `IntStream` for the `values` array's elements, then

- lines 14–15 use the `mapToObj` and `collect` methods to obtain a `String` representation of the stream's elements separated by spaces.

We use this technique several times in this example and subsequent examples to display stream elements.

Streams Cannot Be Reused

This example repeatedly creates an `IntStream` for the `values` array's elements using:

```
IntStream.of(values)
```

You might think that we could simply store the stream and reuse it. However, **once a stream pipeline is processed with a terminal operation, the stream cannot be reused because it does not maintain a copy of the original data source**. Attempting to reuse a stream results in an `IllegalStateException`.

14.7.2 Terminal Operations `count`, `min`, `max`, `sum` and `average`

Class `IntStream` provides various terminal operations for common stream reductions on streams of `int` values:

Click here to view code image

```
17          // count, min, max, sum and average of the values
18          System.out.printf("%nCount: %d%n", IntStream.of(values).count());
19          System.out.printf("Min: %d%n",
20             IntStream.of(values).min().getAsInt());
21          System.out.printf("Max: %d%n",
22             IntStream.of(values).max().getAsInt());
23          System.out.printf("Sum: %d%n", IntStream.of(values).sum());
24          System.out.printf("Average: %.2f%n",
25             IntStream.of(values).average().getAsDouble());
26
```

```
Count: 10
Min: 1
Max: 10
Sum: 55
Average: 5.50
```

Lines 18–25 operate as follows:

- **`count`** (line 18) returns the number of elements in the stream.
- **`min`** (line 20) returns an **`OptionalInt`** (package `java.util`), possibly containing the smallest `int` in the stream. Any stream can be empty. Returning `OptionalInt` enables `min` to return the minimum value if the stream contains at least one element. In this example, we know the stream has 10 elements, so we call the **`OptionalInt`'s `getAsInt` method** to obtain the minimum value. If the stream were empty, the `OptionalInt` would not contain an `int`, and `getAsInt` would throw a `NoSuchElementException`. To prevent this, you can replace the `getAsInt` call with a call to **method `orElse`**, which returns the `OptionalInt`'s value if there is one; otherwise it returns the value you pass to `orElse`. So the following pipeline would return `-1` if the stream were empty:

  Click here to view code image

  ```
  IntStream.of(values).min().orElse(-1)
  ```

- **`max`** (line 22) returns an `OptionalInt`, possibly containing the largest `int` in the stream. Again, we call the `OptionalInt`'s `getAsInt` method to get the largest value because we know this stream contains elements.
- `sum` (line 23) returns the sum of all the `int`s in the stream.
- **`average`** (line 25) returns an **`OptionalDouble`** (package `java.util`), possibly containing the average of the `int`s in the stream as a value of type `double`. In this example, we know the stream has elements, so we call class **`OptionalDouble`'s `getAsDouble` method** to obtain the average. If there were no elements, the `OptionalDouble` would not contain the average, and `getAsDouble` would throw a `NoSuchElementException`. As with `OptionalInt`, to prevent this exception, you can instead call **`OptionalDouble`'s `orElse` method**, which returns the `OptionalDouble`'s value if there is one; otherwise, it returns the value you pass to `orElse`.

Class `IntStream` also provides **method `summaryStatistics`** that performs the `count`, `min`, `max`, `sum` and `average` operations in one pass of an `IntStream`'s elements and returns the results as an **`IntSummaryStatistics`** object (package `java.util`), rather than reprocessing an `IntStream` repeatedly for each operation. This

object has methods for obtaining each result and a `toString` method that summarizes all the results. For example, the statement:

Click here to view code image

```
System.out.println(IntStream.of(values).summaryStatistics());
```

produces:

Click here to view code image

```
IntSummaryStatistics{count=10, sum=55, min=1, average=5.500000,
max=10}
```

for the array `values` in Fig. 14.5.

14.7.3 Terminal Operation `reduce`

So far, we've presented various predefined `IntStream` reductions. You can define your own reductions via **`IntStream`'s `reduce` method**—in fact, each terminal operation discussed in Section 14.7.2 is a specialized implementation of `reduce`.

Click here to view code image

```
27          // sum of values with reduce method
28          System.out.printf("%nSum via reduce method: %d%n",
29             IntStream.of(values)
30                      .reduce(0, (x, y) -> x + y));
31
```

```
Sum via reduce method: 55
```

The stream pipeline in lines 29–30

Click here to view code image

```
IntStream.of(values)
         .reduce(0, (x, y) -> x + y)
```

totals an `IntStream`'s values using `reduce` rather than `sum`:

- The first argument to `reduce` (`0`) is the operation's **identity value**—a value that, when combined with any stream element (using the lambda in the `reduce`'s second argument), produces the element's original value. For example, the identity value is 0 when summing the elements because any

`int` value added to 0 produces the original `int` value. Similarly, when getting the product of the elements, the identity value is 1 because any `int` value multiplied by 1 results in the original `int` value.

- Method `reduce`'s second argument is a method that receives two `int` values representing the left and right operands of a binary operator, performs a calculation with the values and returns the result. The lambda in line 30 adds the values. **A lambda with two or more parameters must enclose them in parentheses.**

The operation specified by a `reduce`'s argument must be associative—that is, the order in which `reduce` applies the operation to the stream's elements must not matter. This is important, because `reduce` is allowed to apply its operation to the stream elements in any order. A non-associative operation could yield different results based on the processing order. For example, subtraction is not an associative operation:

- the expression 7 – (5 – 3) yields 5 whereas
- the expression (7 – 5) – 3 yields –1.

Associative `reduce` operations are critical for parallel streams (Chapter 18) that split operations across multiple cores for better performance.

Evaluating a Custom Reduction

Based on the stream's elements

```
3 10 6 1 4 8 2 5 9 7
```

the reduction's evaluation proceeds as follows:

```
0 + 3 --> 3
3 + 10 --> 13
13 + 6 --> 19
19 + 1 --> 20
20 + 4 --> 24
24 + 8 --> 32
32 + 2 --> 34
34 + 5 --> 39
39 + 9 --> 48
48 + 7 --> 55
```

The first calculation uses the identity value (0) as the left operand, and each subsequent calculation uses the result of the prior calculation as the left operand. The reduction process continues producing a running total of the `IntStream`'s values until they've all been used, at which point the final sum is returned.

Calculating the Product of the Values with Method `reduce`

The stream pipeline in lines 34–35 uses the one-argument version of method `reduce`, which returns an `OptionalInt` that, if the stream has elements, contains the product of the `IntStream`'s values; otherwise, the `OptionalInt` does not contain a result.

Click here to view code image

```
32          // product of values with reduce method
33          System.out.printf("Product via reduce method: %d%n",
34             IntStream.of(values)
35                      .reduce((x, y) -> x * y).getAsInt());
36
```

```
Product via reduce method: 3628800
```

Based on the stream's elements

```
3 10 6 1 4 8 2 5 9 7
```

the reduction's evaluation proceeds as follows:

```
         3 * 10 -->         30
        30 *  6 -->        180
       180 *  1 -->        180
       180 *  4 -->        720
       720 *  8 -->      5,760
     5,760 *  2 -->     11,520
    11,520 *  5 -->     57,600
    57,600 *  9 -->    518,400
   518,400 *  7 --> 3,628,800
```

This process continues producing a running product of the `IntStream`'s values until they've all been used, then the final product is returned.

We could have used the two-parameter `reduce` method, as in:

Click here to view code image

```
IntStream.of(values)
         .reduce(1, (x, y) -> x * y)
```

However, if the stream were empty, this version of `reduce` would return the identity value (`1`), which would not be the expected result for an empty stream.

Summing the Squares of the Values

Now consider summing the squares of the stream's elements.

Click here to view code image

```
37          // sum of squares of values with map and sum methods
38          System.out.printf("Sum of squares via map and sum: %d%n%n",
39             IntStream.of(values)
40                      .map(x -> x * x)
41                      .sum());
42
```

```
Sum of squares via map and sum: 385
```

When implementing your stream pipelines, it's helpful to break down the processing steps into easy-to-understand tasks. Summing the squares of the stream's elements requires two distinct tasks:

- squaring the value of each stream element
- summing the resulting values.

Rather than defining this with a `reduce` method call, the stream pipeline in lines 39–41

```
IntStream.of(values)
         .map(x -> x * x)
         .sum());
```

uses the `map` and `sum` methods to compose the sum-of-squares operation. First, `map` produces a new `IntStream` containing the original element's squares, then `sum` totals the resulting stream's elements.

14.7.4 Sorting `IntStream` Values

In Section 6.19, you sorted arrays using the `sort static` method of

class `Arrays`. You also may sort the elements of a stream.

Click here to view code image

```
43          // displaying the elements in sorted order
44          System.out.printf("Values displayed in sorted order: %s%n",
45             IntStream.of(values)
46                      .sorted()
47                      .mapToObj(String::valueOf)
48                      .collect(Collectors.joining(" ")));
49       }
50    }
```

```
Values displayed in sorted order: 1 2 3 4 5 6 7 8 9 10
```

The stream pipeline in lines 45–48

Click here to view code image

```
IntStream.of(values)
         .sorted()
         .mapToObj(String::valueOf)
         .collect(Collectors.joining(" ")));
```

sorts the stream's elements and displays each value followed by a space. **`IntStream` inter-mediate operation `sorted`** orders the elements of the stream into ascending order by default. Like `filter`, `sorted` is a lazy operation that's performed only when a terminal operation initiates the stream pipeline's processing.

Generative AI

1 Prompt genAIs to discuss and provide brief code examples of the `java.util` package's classes `Optional`, `OptionalInt`, `OptionalLong` and `OptionalDouble`.
2 Prompt genAIs asking when to use the single-argument vs. two-argument versions of `IntStream` method `reduce`.

14.8 Functional Interfaces

[This section requires the interface concepts introduced in

Sections 9.11–9.12.]

Section 9.12 introduced `default`, `static` and `private` interface methods. It also discussed the concept of a functional interface containing exactly one `abstract` method (and possibly `default`, `static` and `private` methods). Such interfaces are also known as single abstract method (SAM) interfaces. Functional interfaces are used extensively in functional-style Java programming. Functional programmers work with **pure functions** that depend only on the argument(s) they receive. Given a particular argument (or arguments), a pure function always produces the same result. Pure functions have **referential transparency**—that is, they:

- depend only on their parameters,
- have no side effects, and
- do not maintain any state.

State changes occur by passing data from method to method. No data is shared. Pure functions are safer because they do not modify a program's state (variables). This also makes them less error-prone and thus easier to test, modify and debug, and in the context of stream pipelines, easier to parallelize.

Functional Interfaces in Package `java.util.function`

Package `java.util.function` contains several functional interfaces. The following table contains the six basic generic functional interfaces, several of which you've already used in this chapter's examples. Throughout the table, `T` and `R` are generic type names that represent the type of object on which the functional interface operates and the method's return type, respectively.

Interface	Description
`BinaryOperator<T>`	Represents a method that takes two parameters of the same type and returns a value of that type. Performs a task using the parameters (such as a calculation) and returns the result. The lambdas you passed to `IntStream` method `reduce` (Section 14.7) implemented `IntBinaryOperator`—an `int`-specific version of `BinaryOperator`.
`Consumer<T>`	Represents a one-parameter method that returns `void`. Performs a task using its parameter, such as outputting the object, invoking a method of the object, etc. The lambda you passed to `IntStream` method `forEach` (Section 14.6) implemented interface `IntConsumer`—an `int`-specific version of `Consumer`. Later sections

present several more examples of `Consumer`s.

`Function<T,R>` Represents a one-parameter method that performs a task on the parameter and returns a result—possibly of a different type than the parameter. The lambda you passed to `IntStream` method `mapToObj` (Section 14.6) implemented interface `IntFunction`—an `int`-specific version of `Function`. Later sections present several more examples of `Function`s.

`Predicate<T>` Represents a one-parameter method that returns a `boolean` result. Determines whether the parameter satisfies a condition. The lambda you passed to `IntStream` method `filter` (Section 14.4) implemented interface `IntPredicate`—an `int`-specific version of `Predicate`. Later sections present several more examples of `Predicate`s.

`Supplier<T>` Represents a no-parameter method that returns a result. Often used to create a collection object in which a stream operation's results are placed. We'll demonstrate `Supplier`s starting in Section 14.13.

`UnaryOperator<T>` Represents a one-parameter method that returns a result of the same type as its parameter. The lambdas you passed in Section 14.3 to `IntStream` method `map` implemented `IntUnaryOperator`—an `int`-specific version of `UnaryOperator`. Later sections present more examples of `UnaryOperator`s.

Many other functional interfaces in package `java.util.function` are specialized versions of those in the table above. Most are for use with `int`, `long` and `double` primitive values. There are also generic customizations of `Consumer`, `Function` and `Predicate` for binary operations—methods that take two arguments. For each `IntStream` method we've used that receives a lambda, the method's parameter is actually an `int`-specific version of one of these interfaces.

Generative AI

1 Prompt genAIs asking for a list of commonly used Java interfaces outside the package `java.util.function` that can be implemented with lambdas.

14.9 Lambdas: A Deeper Look

Type Inference and a Lambda's Target Type

Lambda expressions can be used anywhere functional interfaces are expected. The Java compiler can usually infer the types of a lambda's parameters and the type returned by a lambda from the context in which the lambda is used. This is determined by the **lambda's target type**—the functional interface type that's expected where the lambda appears in the code. For example, in the call to `IntStream` method `map` from Fig. 14.2

Click here to view code image

```
IntStream.rangeClosed(1, 10)
         .map((int x) -> {return x * 2;})
         .sum()
```

the target type is `IntUnaryOperator`—a method that takes one `int` parameter and returns an `int` result. Here, the lambda parameter's type is explicitly declared as `int`. The compiler infers the lambda's return type as `int` because that's what an `IntUnaryOperator` requires.

The compiler also can infer a lambda parameter's type. For example, in the call to `IntStream` method `filter` from stream pipeline in Fig. 14.3

Click here to view code image

```
IntStream.rangeClosed(1, 10)
         .filter(x -> x % 2 == 0)
         .map(x -> x * 2)
         .sum()
```

the target type is `IntPredicate`—a method that takes one `int` parameter and returns a `boolean` result. The compiler infers the lambda parameter `x`'s type as `int` because that's what an `IntPredicate` requires. We generally let the compiler infer the lambda parameter's type in our examples.

Scope and Lambdas

Unlike methods, lambdas do not have their own scope. So, for example, you cannot shadow an enclosing method's local variables with lambda parameters having the same names. A compilation error occurs in this case, because the method's local variables and the lambda parameters are in the same scope. In a lambda with its body in braces, any variables declared in that body have local

scope.

Capturing Lambdas and `final` Local Variables

A lambda that refers to a local variable from the enclosing method (that is, the lambda's lexical scope) is a **capturing lambda**. For such a lambda, the compiler captures the local variable's value and stores it with the lambda to ensure that the lambda can use the value when the lambda eventually executes. This is important because you can pass a lambda to another method that executes the lambda after its lexical scope no longer exists.

Any local variable a lambda references in its lexical scope must be `final`. Such a variable can be explicitly declared `final` or effectively `final`. For an effectively `final` variable, the compiler infers that the local variable could have been declared `final` because its enclosing method never modifies the variable after it's declared and initialized.

Generative AI

1 Prompt genAIs, asking why Java lambda expressions do not have their own scope.
2 Prompt genAIs, asking for code examples demonstrating capturing lambdas in Java.

14.10 `Stream<Integer>` Manipulations

[This section requires the interface concepts introduced in Sections 9.11–9.12.]
So far, we've processed `IntStream`s. **Interface `Stream`** (package `java.util.stream`) is a generic interface for performing stream operations on reference-type objects. The `Stream`'s source determines the types of objects processed. `IntStream` is simply an `int`-optimized `Stream` that provides methods for common `int` operations. Figure 14.6 performs filtering and sorting on a `Stream<Integer>`, using techniques similar to prior examples, and places a stream pipeline's results into a new collection for subsequent processing. We'll work with `Stream`s of other reference types in subsequent examples. We split this example's code into parts that we discuss in Sections 14.10.1–14.10.4.

Click here to view code image

```
1  // Fig. 14.6: ArraysAndStreams.java
2  // Demonstrating lambdas and streams with an array of Integers.
3  import java.util.Arrays;
4  import java.util.List;
5  import java.util.stream.Collectors;
6
7  public class ArraysAndStreams {
8     public static void main(String[] args) {
9        Integer[] values = {2, 9, 5, 0, 3, 7, 1, 4, 8, 6};
10
11       // display original values
12       System.out.printf("Original values: %s%n", Arrays.asList(values));
13
```

```
Original values: [2, 9, 5, 0, 3, 7, 1, 4, 8, 6]
```

Fig. 14.6 | Demonstrating lambdas and streams with an array of `Integer`s.

Throughout this example, we use the `Integer` array `values` (line 9) initialized with `int` values—the compiler boxes each `int` into an `Integer` object. Line 12 displays the con-tents of `values` before we perform any stream processing. `Arrays` method `asList` creates a `List<Integer>` view of the `values` array. The generic interface `List` is implemented by collections like `ArrayList` (Chapter 6). Line 12 displays the `List<Integer>`'s default `String` representation, which consists of square brackets (`[` and `]`) containing a comma-separated list of elements—we use this `String` representation throughout the example.

14.10.1 Creating a `Stream<Integer>`, Sorting It and Collecting the Results

Class `Arrays` `stream` method creates a `Stream` from an array of objects—for example, line 16 produces a `Stream<Integer>` because `stream`'s argument is an array of `Integer`s.

Click here to view code image

```
14       // sort values in ascending order with streams
15       System.out.printf("Sorted values: %s%n",
16          Arrays.stream(values)
17                .sorted()
18                .collect(Collectors.toList()));
19
```

```
Sorted values: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Class `Arrays` also provides overloaded versions of method `stream` to create `IntStream`s, `LongStream`s and `DoubleStream`s from entire `int`, `long` and `double` arrays or subsets of their elements. The stream pipeline in lines 16–18

Click here to view code image

```
Arrays.stream(values)
      .sorted()
      .collect(Collectors.toList())
```

sorts the `values` array's elements (without modifying the original array) and collects the results in a `List<Integer>`. First, line 16 creates a `Stream<Integer>` from `values`. Next, line 17 calls `Stream` method `sorted` to sort the elements—this results in an intermediate `Stream<Integer>` with the values in ascending order. (Section 14.11.3 discusses how to sort in descending order.)

Creating a New Collection Containing a Stream Pipeline's Results

When processing streams, you often create new collections containing the results so that you can perform operations on them later. To do so, you can use **`Stream`'s terminal operation `collect`** (Fig. 14.6, line 18). As the stream pipeline is processed, method `collect` performs a **mutable reduction operation** that creates a mutable `List`, `Map` or `Set`, modifying it by adding the pipeline's results to the collection. Similarly, the **mutable reduction operation `toArray`** places the results in a new mutable array of the `Stream`'s element type.

The version of method `collect` in line 18 receives as its argument an object that implements **interface `Collector`** (package `java.util.stream`), which specifies how to perform the mutable reduction. Class `Collectors` (package `java.util.stream`) provides `static` methods that return predefined `Collector` implementations. `Collectors` method **`toList`** (line 18) returns a `Collector` that places the `Stream<Integer>`'s elements into a mutable `List<Integer>` collection. In lines 15–18, the resulting `List<Integer>` is displayed with an implicit call to its `toString` method. If you know that you will not modify the `List`, you can replace line 18 with a call to **`Stream` method `toList`**:

```
.toList()
```

This returns an immutable object that implements the `List` interface. We'll use this approach in subsequent examples.

A mutable reduction optionally performs a final data transformation. For example, in Fig. 14.4, we called `Stream` method `collect` with the object returned by `Collectors` method `joining`. Behind the scenes, this `Collector` used a **`StringJoiner`** (package `java.util`) to concatenate the stream elements' `String` representations, then called the `StringJoiner`'s `toString` method to transform the result into a `String`. We present additional `Collectors` in Section 14.12. For more predefined `Collectors`, visit:

Click here to view code image

```
https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/
   util/stream/Collectors.html
```

14.10.2 Filtering a `Stream` and Storing the Results for Later Use

The pipeline in lines 21–24 creates a `Stream<Integer>`, `filter`s the stream to locate all the values greater than 4 and collects the results into an immutable `List<Integer>`. `Stream` method `filter`'s lambda argument implements the functional interface `Predicate` (package `java.util.function`), representing a one-parameter method that returns a `boolean` indicating whether the parameter value satisfies the predicate. We stored the stream pipe-line's resulting `List<Integer>` in `greaterThan4`, which we use in line 25 to display the values greater than 4. We'll use `greaterThan4` again in lines 37–39 to perform additional operations on the values greater than 4.

Click here to view code image

```
20          // values greater than 4
21          List<Integer> greaterThan4 =
22             Arrays.stream(values)
23                   .filter(value -> value > 4)
24                   .toList());
25          System.out.printf("Values greater than 4: %s%n", greaterThan4);
26
```

```
Values greater than 4: [9, 5, 7, 8, 6]
```

14.10.3 Filtering and Sorting a `Stream` and Collecting the Results

The stream pipeline in lines 29–32 displays the values greater than 4 in sorted order. First, line 29 creates a `Stream<Integer>`. Then, line 30 `filter`s the elements to locate all the values greater than 4. Next, line 31 indicates that we'd like the results `sorted`. Finally, line 32 collects the results into a `List<Integer>`, which is then displayed by `printf` as a `String`. We called `filter` before `sorted` so only the elements that will be in the pipeline's result get sorted.

Click here to view code image

```
27          // filter values greater than 4 then sort the results
28          System.out.printf("Sorted values greater than 4: %s%n",
29             Arrays.stream(values)
30                   .filter(value -> value > 4)
31                   .sorted()
32                   .toList());
33
```

```
Sorted values greater than 4: [5, 6, 7, 8, 9]
```

14.10.4 Sorting Previously Collected Results

The stream pipeline in lines 37–39 uses the `greaterThan4` collection created in lines 21–24 to demonstrate additional processing on the results of a prior stream pipeline. `List` **method** **`stream`** creates the stream, then we sort the elements and collect the results into a new `List<Integer>` and display its `String` representation.

Click here to view code image

```
34          // greaterThan4 List sorted with streams
35          System.out.printf(
36             "Values greater than 4 (ascending with streams): %s%n",
37             greaterThan4.stream()
38                   .sorted()
39                   .toList());
40       }
41    }
```

```
Values greater than 4 (ascending with streams): [5, 6, 7, 8, 9]
```

Generative AI

1 Prompt genAIs, asking them to explain the difference between Java's mutable and immutable reduction operations.
2 Prompt genAIs, asking them to explain the difference between Java's `IntStream` and `Stream<Integer>` and when each is preferred.

14.11 `Stream<String>` Manipulations

[This section demonstrates how lambdas and streams can simplify programming tasks you learned in Chapter 7, Strings, NLP and Regex: Generative AI Foundations.]

So far, we've manipulated only streams of `int` values and `Integer` objects. Figure 14.7 performs stream operations on a `Stream<String>`. In addition, we demonstrate case-insensitive sorting and sorting in descending order. Throughout this example, we use the `String` array `strings` (lines 7–8) that's initialized with color names—some with an initial uppercase letter. Line 11 displays the contents of `strings` before we perform any stream processing. We split the code into parts discussed in Sections 14.11.1–14.11.3.

Click here to view code image

```
1  // Fig. 14.7: ArraysAndStreams2.java
2  // Demonstrating lambdas and streams with an array of Strings.
3  import java.util.Arrays;
4
5  public class ArraysAndStreams2 {
6     public static void main(String[] args) {
7        String[] strings =
8           {"Red", "orange", "Yellow", "green", "Blue", "indigo", "Violet"};
9
10       // display original strings
11       System.out.printf("Original strings: %s%n", Arrays.asList(strings));
12
```

```
Original strings: [Red, orange, Yellow, green, Blue, indigo, Violet]
```

Fig. 14.7 | Demonstrating lambdas and streams with an array of `String`s.

14.11.1 Mapping `String`s to Uppercase

The stream pipeline in lines 15–17 displays the `String`s in uppercase letters:

Click here to view code image

```
13          // strings in uppercase
14          System.out.printf("strings in uppercase: %s%n",
15             Arrays.stream(strings)
16                   .map(String::toUpperCase)
17                   .toList());
18
```

```
strings in uppercase: [RED, ORANGE, YELLOW, GREEN, BLUE, INDIGO, VIOLET]
```

Line 15 creates a `Stream<String>` from the array `strings`, then line 16 `map`s each `String` to its uppercase version by calling `String` instance method `toUpperCase` on each stream element. `Stream` method `map` receives an object that implements the `Function` functional interface, representing a one-parameter method that performs a task with its parameter then returns the result. In this case, we pass to `map` an **unbound instance method reference** of the form *ClassName*`::`*instanceMethodName* (`String::toUpperCase`). "Unbound" means that the method reference does not indicate the specific object on which the method will be called—the compiler converts this to a one-parameter lambda that invokes the instance method on the lambda's parameter, which must have type *ClassName*. In this case, the compiler converts `String::toUpperCase` into a lambda like

```
s -> s.toUpperCase()
```

which returns the uppercase version of the lambda's argument. Line 17 `collect`s the results into a `List<String>` that we output as a `String`.

14.11.2 Filtering `String`s Then Sorting Them in Case-Insensitive Ascending Order

The stream pipeline in lines 21–24 filters and sorts the `String`s:

Click here to view code image

```
19          // strings less than "n" (case insensitive) sorted ascending
20          System.out.printf("strings less than n sorted ascending: %s%n",
```

```
21            Arrays.stream(strings)
22                  .filter(s -> s.compareToIgnoreCase("n") < 0)
23                  .sorted(String.CASE_INSENSITIVE_ORDER)
24                  .toList());
25
```

```
strings less than n sorted ascending: [Blue, green, indigo]
```

Line 21 creates a `Stream<String>` from the array `strings`, then line 22 calls `Stream` method `filter` to locate all the `String`s that are less than `"n"`, using a case-insensitive comparison in the `Predicate` lambda. Line 23 sorts the results, and line 24 collects them into a `List<String>` that we output as a `String`.

In this case, line 23 invokes the `Stream` method `sorted` that receives a `Comparator` (Section 12.6.1) as an argument. By default, method `sorted` uses the natural order for the type—for `String`s, the natural order is case sensitive (lexicographical), which means that `"Z"` is less than `"a"`. Passing the predefined `Comparator` `String.CASE_INSENSITIVE_ORDER` per-forms a case-insensitive sort.

14.11.3 Filtering `String`s Then Sorting Them in Case-Insensitive Descending Order

The stream pipeline in lines 28–31 performs the same tasks as lines 21–24 but sorts the `String`s in descending order. Functional interface `Comparator` contains `default` method **`reversed`**, which reverses an existing `Comparator`'s ordering. When you apply `reversed` to `String.CASE_INSENSITIVE_ORDER`, `sorted` performs a case-insensitive sort and places the `String`s in descending order.

Click here to view code image

```
26          // strings less than "n" (case insensitive) sorted descending
27          System.out.printf("strings less than n sorted descending: %s%n",
28             Arrays.stream(strings)
29                  .filter(s -> s.compareToIgnoreCase("n") < 0)
30                  .sorted(String.CASE_INSENSITIVE_ORDER.reversed())
31                  .toList());
32       }
33    }
```

```
strings less than n sorted descending: [indigo, green, Blue]
```

Generative AI

1 Prompt genAIs to write a Java stream pipeline that finds the longest `String` in a `List<String>`.

14.12 `Stream<Employee>` Manipulations

[This section demonstrates how lambdas and streams can be used to simplify programming tasks that you learned in Chapter 12, Generic Collections.]

The previous examples in this chapter performed stream manipulations on primitive types (like `int`) and Java class library types (like `Integer` and `String`). Of course, you also may perform operations on collections of programmer-defined types.

The example in Figs. 14.8–14.9 demonstrates various lambda and stream capabilities using a `Stream<Employee>`. The `record` class `Employee` (Fig. 14.8) represents an employee with a first name, last name, salary and department. The `getName` method (lines 6–8) returns the combined first and last name as a `String`, and the `toString` method (lines 11–15) returns a formatted `String` containing the employee's first name, last name, salary and department. We discuss Fig. 14.9 in Sections 14.12.1–14.12.7.

Click here to view code image

```
1  // Fig. 14.8: Employee.java
2  // Employee class.
3  public record Employee(
4     String firstName, String lastName, double salary, String department) {
5     // return Employee's first and last name combined
6     public String getName() {
7        return String.format("%s %s", firstName(), lastName());
8     }
9
10    // return a String containing the Employee's information
11    @Override
12    public String toString() {
13       return String.format("%-8s %-8s %8.2f %s",
14          firstName(), lastName(), salary(), department());
15    }
16 }
```

Fig. 14.8 | `Employee` class for use in Fig. 14.9.

14.12.1 Creating and Displaying a `List<Employee>`

We broke class `ProcessingEmployees` (Fig. 14.9) into parts for discussion purposes. Lines 14–21 create an immutable `List<Employee>` using `List` `static` method `of` (Section 12.11).

Click here to view code image

```
1  // Fig. 14.9: ProcessingEmployees.java
2  // Processing streams of Employee objects.
3  import java.util.Comparator;
4  import java.util.List;
5  import java.util.Map;
6  import java.util.function.Function;
7  import java.util.function.Predicate;
8  import java.util.stream.Collectors;
9
10 public class ProcessingEmployees {
11    public static void main(String[] args) {
12
13       // initialize List<Employee>
14       List<Employee> list = List.of(
15          new Employee("James", "Roy", 5000, "IT"),
16          new Employee("Zaynab", "Hanna", 7600, "IT"),
17          new Employee("Zoe", "Lopez", 3587.5, "Sales"),
18          new Employee("Mateo", "Lopez", 4700.77, "Marketing"),
19          new Employee("Diego", "Lopez", 3200, "IT"),
20          new Employee("Emma", "Bernard", 6200, "Sales"),
21          new Employee("Xinyi", "Chen", 4236.4, "Marketing"));
22
```

Fig. 14.9 | Processing streams of `Employee` objects.

Printing the `Employee`s

Line 25 creates a `Stream<Employee>`, then uses `Stream` method `forEach` to display each `Employee`'s `String` representation:

Click here to view code image

```
23       // display all Employees
24       System.out.println("Complete Employee list:");
25       list.stream().forEach(System.out::println);
26
```

```
Complete Employee list:
```

```
James     Roy        5000.00   IT
Zaynab    Hanna      7600.00   IT
Zoe       Lopez      3587.50   Sales
Mateo     Lopez      4700.77   Marketing
Diego     Lopez      3200.00   IT
Emma      Bernard    2200.00   Sales
Xinyi     Chen       4236.40   Marketing
```

`Stream` method `forEach` expects as its argument an object that implements the `Consumer` functional interface, which represents an action to perform on each stream element—the corresponding method receives one argument and returns `void`. The bound instance method reference `System.out::println` is converted by the compiler into a one-parameter lambda that passes the lambda's argument—an `Employee`—to the `System.out` object's `println` instance method, which implicitly calls class `Employee`'s `toString` method to get the `String` representation. In line 25, `Stream` method `forEach` passes each `Employee` to the `System.out` object's `println` method, which calls the `Employee`'s `toString` method.

14.12.2 Filtering `Employee`s with Salaries in a Specified Range

So far, we've used lambdas only by passing them directly as arguments to stream methods. Here, we store a lambda in a variable for later use. Lines 28–29 declare a variable of the functional interface type `Predicate<Employee>` and initialize it with a one-parameter lambda that returns a `boolean` (as `Predicate` requires). The lambda returns `true` if an `Employee`'s salary range from 4000 to 6000. We use the stored lambda in lines 36 and 43 to filter `Employee`s.

Click here to view code image

```
27          // Predicate that returns true for salaries in the range $4000-$6000
28          Predicate<Employee> salaryInRange4to6k =
29             e -> (e.salary() >= 4000 && e.salary() <= 6000);
30
31          // Display Employees with salaries in the range $4000-$6000
32          // sorted into ascending order by salary
33          System.out.printf(
34             "%nEmployees earning $4000-$6000 per month sorted by salary:%n");
35          list.stream()
36              .filter(salaryInRange4to6k)
37              .sorted(Comparator.comparing(Employee::salary))
```

```
38              .forEach(System.out::println);
39
```

```
Employees earning $4000-$6000 per month sorted by salary:
Xinyi    Chen      4236.40   Marketing
Mateo    Lopez     4700.77   Marketing
James    Roy       5000.00   IT
```

The stream pipeline in lines 35–38 performs the following tasks:

- Line 35 creates a `Stream<Employee>`.
- Line 36 filters the stream using the `Predicate` named `salaryInRange4to6k`.
- Line 37 sorts remaining `Employees`. To create a salary `Comparator`, we use interface `Comparator`'s `static` method `comparing`, which receives a `Function` that performs a task on its argument and returns the result. The compiler converts the unbound instance method reference `Employee::salary` into a one-parameter lambda that calls `salary` on its `Employee` argument. The `Comparator` returned by method `comparing` calls its `Function` argument on each of two `Employee` objects, then returns a negative value if the first `Employee`'s salary is less than the second, 0 if they're equal and a positive value if the first `Employee`'s salary is greater than the second. `Stream` method `sorted` uses these values to order the `Employee`s.
- Finally, line 38 performs the terminal `forEach` operation that processes the stream pipeline and outputs the `Employee`s sorted by salary.

Short-Circuit Stream Pipeline Processing

Section 4.10 discussed short-circuit evaluation with the logical AND (`&&`) and logical OR (`||`) operators. A nice performance feature of lazy evaluation is the ability to perform short-circuit evaluation—that is, to stop processing the stream pipeline as soon as the desired result is available. Line 44 demonstrates **`Stream` method `findFirst`**—a short-circuiting terminal operation that processes the stream pipeline and terminates processing as soon as the first object from the stream's intermediate operation(s) is found.

Click here to view code image

```
40          // Display first Employee with salary in the range $4000-$6000
41          System.out.printf("%nFirst employee who earns $4000-$6000:%n%s%n",
42             list.stream()
43                 .filter(salaryInRange4to6k)
44                 .findFirst()
45                 .get());
46
```

```
First employee who earns $4000-$6000:
James     Roy        5000.00    IT
```

The stream pipeline in lines 42–45, which filters `Employee`s with salaries in the range $4000–$6000, proceeds as follows:

- The `Predicate salaryInRange4to6k` is applied to the first `Employee` (James Roy). His salary ($5000.00) is in the range $4000–$6000, so the `Predicate` returns `true`, and processing of the stream terminates immediately, having processed only one of the eight objects in the stream.
- Method `findFirst` then returns an `Optional` (an `Optional<Employee>` in this case) containing the object found, if any. The call to `Optional` method `get` (line 45) returns the matching `Employee` object in this example. Even if the stream contained millions of `Employee` objects, the `filter` operation would be performed only until a match was found.

We knew from this example's `Employee`s that this pipeline would find at least one `Employee` with a salary in the range 4000–6000. So, we called `Optional` method `get` without first checking whether the `Optional` contained a result. If `findFirst` yields an empty `Optional`, this would cause a `NoSuchElementException`. For a stream operation that returns an `Optional<T>`, you can store the result in a variable, then use its `isPresent` or `ifPresent` methods to confirm whether there's a result before attempting to use it. This prevents `NoSuchElementException`s.

Method `findFirst` is one of several search-related terminal operations. The following table contains several similar `Stream` methods.

Search-related terminal stream operations

`findAny`	Similar to `findFirst` but finds and returns any stream element based on the prior intermediate operations. Immediately terminates processing of the stream pipeline once such an

element is found. Typically, `findFirst` is used with sequential streams, and `findAny` is used with parallel streams (Section 18.3).

`anyMatch` Determines whether any stream elements match a specified condition. Returns `true` if at least one stream element matches and `false` otherwise. Immediately terminates processing of the stream pipeline if an element matches.

`allMatch` Determines whether all of the elements in the stream match a specified condition. Returns `true` if so and `false` otherwise. Immediately terminates processing of the stream pipeline if any element does not match.

14.12.3 Sorting `Employee`s By Multiple Fields

Next, let's use streams to sort objects by multiple fields. We'll sort `Employee`s by last name, then for `Employee`s with the same last name, we also sort them by first name. To do so, we begin by creating two `Function`s that each receive an `Employee` and return a `String`:

- `byFirstName` (line 48) is assigned a method reference for `Employee` instance method `firstName`
- `byLastName` (line 49) is assigned a method reference for `Employee` instance method `lastName`

Next, we use these `Function`s to create a `Comparator` (`lastThenFirst`; lines 52–53) that compares two `Employee`s by last name and, if necessary, by first name. We use `Comparator` method `comparing` to create a `Comparator` that calls `Function byLastName` on an `Employee` to get its last name. On the resulting `Comparator`, we call **`Comparator` method `thenComparing`** to create a composed `Comparator` that first compares `Employee`s by last name and, if the last names are equal, then compares them by first name.

Click here to view code image

```
47          // Functions for getting first and last names from an Employee
48          Function<Employee, String> byFirstName = Employee::firstName;
49          Function<Employee, String> byLastName = Employee::lastName;
50
51          // Comparator for comparing Employees by first name then last name
52          Comparator<Employee> lastThenFirst =
53             Comparator.comparing(byLastName).thenComparing(byFirstName);
54
```

```
55          // sort employees by last name, then first name
56          System.out.printf(
57             "%nEmployees in ascending order by last name then first:%n");
58          list.stream()
59              .sorted(lastThenFirst)
60              .forEach(System.out::println);
61
62          // sort employees in descending order by last name, then first name
63          System.out.printf(
64             "%nEmployees in descending order by last name then first:%n");
65          list.stream()
66              .sorted(lastThenFirst.reversed())
67              .forEach(System.out::println);
68
```

```
Employees in ascending order by last name then first:
Emma      Bernard    6200.00   Sales
Xinyi     Chen       4236.40   Marketing
Zaynab    Hanna      7600.00   IT
Diego     Lopez      3200.00   IT
Mateo     Lopez      4700.77   Marketing
Zoe       Lopez      3587.50   Sales
James     Roy        5000.00   IT

Employees in descending order by last name then first:
James     Roy        5000.00   IT
Zoe       Lopez      3587.50   Sales
Mateo     Lopez      4700.77   Marketing
Diego     Lopez      3200.00   IT
Zaynab    Hanna      7600.00   IT
Xinyi     Chen       4236.40   Marketing
Emma      Bernard    6200.00   Sales
```

Lines 58–60 use this new `lastThenFirst Comparator` to sort the `Employee`s in ascending order then display the results. We reuse the `Comparator` in lines 65–67 but call its `reversed` method to indicate that the `Employee`s should be sorted in descending order by last name, then first name. Lines 48–53 may be expressed more concisely as:

Click here to view code image

```
Comparator<Employee> lastThenFirst =
   Comparator.comparing(Employee::lastName)
             .thenComparing(Employee::firstName);
```

Aside: Composing Lambda Expressions

Many functional interfaces in the package `java.util.function` package provide `default` methods for composing functionality. For example, consider the interface `IntPredicate`, which contains three `default` methods:

- **`and`**—performs a logical AND with short-circuit evaluation between the `IntPredicate` on which it's called and the `IntPredicate` it receives as an argument.
- **`negate`**—reverses the `boolean` value of the `IntPredicate` on which it's called.
- **`or`**—performs a logical OR with short-circuit evaluation between the `IntPredicate` on which it's called and the `IntPredicate` it receives as an argument.

You can use these methods and `IntPredicate` objects to compose more complex conditions. For example, consider the following two `IntPredicate`s that are each initialized with lambdas:

Click here to view code image

```
IntPredicate even = value -> value % 2 == 0;
IntPredicate greaterThan5 = value -> value > 5;
```

To locate all the even integers greater than 5 in an `IntStream`, you could pass to `IntStream` method `filter` the following composed `IntPredicate`:

```
even.and(greaterThan5)
```

Like `IntPredicate`, functional interface `Predicate` represents a method that returns a `boolean` indicating whether its argument satisfies a condition. `Predicate` also contains methods **`and`** and **`or`** for combining predicates, and **`negate`** for reversing a predicate's `boolean` value.

14.12.4 Mapping `Employee`s to Unique-Last-Name `String`s

You previously used `map` operations to perform calculations on `int` values, convert `int`s to `String`s and convert `String`s to uppercase letters. Next, we map objects of one type (`Employee`) to objects of a different type (`String`).

Click here to view code image

```
69          // display unique employee last names sorted
70          System.out.printf("%nUnique employee last names:%n");
71          list.stream()
72              .map(Employee::lastName)
73              .distinct()
74              .sorted()
75              .forEach(System.out::println);
76
77          // display only first and last names
78          System.out.printf(
79             "%nEmployee names in order by last name then first name:%n");
80          list.stream()
81              .sorted(lastThenFirst)
82              .map(Employee::getName)
83              .forEach(System.out::println);
84
```

```
Unique employee last names:
Bernard
Chen
Hanna
Lopez
Roy

Employee names in order by last name then first name:
Emma Bernard
Xinyi Chen
Zaynab Hanna
Diego Lopez
Mateo Lopez
Zoe Lopez
James Roy
```

The stream pipeline in lines 71–75 performs the following tasks:

- Line 71 creates a `Stream<Employee>`.
- Line 72 maps the `Employees` to their last names using the unbound instance-method reference `Employee::getName` as `map`'s `Function` argument. The result is a `Stream<String>` containing only the `Employee`s' last names.
- Line 73 calls **`Stream` method `distinct`** on the `Stream<String>` to eliminate any duplicate `String`s—the resulting stream contains only unique last names.
- Line 74 sorts the unique last names.

- Finally, line 75 performs a terminal `forEach` operation that processes the stream pipeline and outputs the unique last names in sorted order.

Lines 80–83 sort the `Employees` by last name, then first name, then `map` the `Employee`s to `String`s with `Employee` instance method `getName` (line 82) and display the sorted names in a terminal `forEach` operation.

14.12.5 Grouping `Employee`s By Department

Previously, we've used the terminal stream operation `collect` to concatenate stream elements into a `String` representation and to place stream elements into `List` collections. Line 89 uses `Stream` method `collect` to group `Employees` by department.

Click here to view code image

```
85          // group Employees by department
86          System.out.printf("%nEmployees by department:%n");
87          Map<String, List<Employee>> groupedByDepartment =
88             list.stream()
89                 .collect(Collectors.groupingBy(Employee::department));
90          groupedByDepartment.forEach(
91             (department, employeesInDepartment) -> {
92                System.out.printf("%n%s%n", department);
93                employeesInDepartment.forEach(
94                   employee -> System.out.printf("   %s%n", employee));
95             }
96          );
97
```

```
Employees by department:

Sales
   Zoe      Lopez     3587.50   Sales
   Emma     Bernard   6200.00   Sales

IT
   James    Roy       5000.00    IT
   Zaynab   Hanna     7600.00    IT
   Diego    Lopez     3200.00    IT

Marketing
   Mateo    Lopez     4700.77    Marketing
   Xinyi    Chen      4236.40    Marketing
```

Recall that `collect`'s argument is a `Collector` that specifies how to summarize the data into a useful form. In this case, we use the `Collector` returned by **`Collectors` `static` method `groupingBy`**, which receives a `Function` that classifies the objects in the stream. The values this `Function` returns are used as the keys in a `Map` collection. The corresponding values, by default, are `List`s containing the stream elements in a given category.

When method `collect` is used with this `Collector`, the result is a `Map<String, List<Employee>>` in which each `String` key is a department and each `List<Employee>` contains the `Employee`s in that department. We assign this `Map` to `groupedByDepartment`, which we use in lines 90–96 to display the `Employees` grouped by department. **`Map` method `forEach`** performs an operation on each of the `Map`'s key–value pairs—in this case, each key is a department, and the corresponding value is a `List<Employee>` containing that department's `Employee`s. The argument to a `Map`'s `forEach` method is an object that implements **functional interface `BiConsumer`**, representing a two-parameter method that does not return a result. For a `Map`, the `BiConsumer`'s first parameter is the key, and the second is the corresponding value. In the lambda's body, line 92 displays the department (i.e., the key). Lines 93–94 use the corresponding `List`'s `forEach` method to output each `Employee`'s `String` representation preceded by three spaces.

14.12.6 Counting the Number of `Employee`s in Each Department

Next, we use `Stream` method `collect` and `Collectors` `static` method `groupingBy` to count the number of `Employee`s in each department. The technique demonstrated combines grouping and reduction into a single operation. The stream pipeline in lines 100–103 produces a `Map<String, Long>` in which each `String` key is a department name, and the corresponding `Long` value is the department's number of `Employee`s.

Click here to view code image

```
98          // count number of Employees in each department
99          System.out.printf("%nCount of Employees by department:%n");
100         Map<String, Long> employeeCountByDepartment =
101            list.stream()
102                .collect(Collectors.groupingBy(Employee::department,
103                   Collectors.counting()));
104         employeeCountByDepartment.forEach(
```

```
105            (department, count) -> System.out.printf(
106               "%s has %d employee(s)%n", department, count));
107
```

```
Count of Employees by department:
Sales has 2 employee(s)
IT has 3 employee(s)
Marketing has 2 employee(s)
```

In this case, we use a version of `Collectors` `static` method `groupingBy` that receives two arguments:

- the first is a `Function` that classifies the objects in the stream and
- the second is another `Collector` (known as the **downstream `Collector`**) used to collect the objects classified by the `Function`.

We use a call to `Collectors` `static` method `counting` as the second argument. This resulting `Collector` reduces the elements in a given classification to a count of those elements, rather than collecting them into a `List`. Lines 104–106 output the key–value pairs from the resulting `Map<String, Long>`.

14.12.7 Summing and Averaging `Employee` Salaries

Previously, we mapped streams of primitive-type elements to streams of objects with method `mapToObj` (found in classes `IntStream`, `LongStream` and `DoubleStream`). Similarly, a `Stream` of objects may be mapped to an `IntStream`, `LongStream` or `DoubleStream`. Here, we use **`Stream` method `mapToDouble`** (lines 112, 119 and 125) to get a `DoubleStream` that maps objects to `double` values. In this case, we map `Employee` objects to their salaries so that we can calculate the sum and average.

Click here to view code image

```
108        // sum of Employee salaries with DoubleStream sum method
109        System.out.printf(
110           "%nSum of Employees' salaries (via sum method): %.2f%n",
111           list.stream()
112               .mapToDouble(Employee::salary)
113               .sum());
```

```
114
115         // calculate sum of Employee salaries with Stream reduce method
116         System.out.printf(
117            "Sum of Employees' salaries (via reduce method): %.2f%n",
118            list.stream()
119                .mapToDouble(Employee::salary)
120                .reduce(0, (value1, value2) -> value1 + value2));
121
122         // average of Employee salaries with DoubleStream average method
123         System.out.printf("Average of Employees' salaries: %.2f%n",
124            list.stream()
125                .mapToDouble(Employee::salary)
126                .average()
127                .getAsDouble());
128      }
129   }
```

```
Sum of Employees' salaries (via sum method): 34524.67
Sum of Employees' salaries (via reduce method): 34524.67
Average of Employees' salaries: 4932.10
```

Method `mapToDouble` receives an object that implements the **functional interface** `ToDoubleFunction` (package `java.util.function`), representing a one-parameter method that returns a `double` value. Lines 112, 119 and 125 each pass to `mapToDouble` the unbound instance-method reference `Employee::salary`, which returns the current `Employee`'s salary as a `double`. The compiler converts this method reference into a one-parameter lambda that calls `salary` on its `Employee` argument.

Lines 111–113 create a `Stream<Employee>`, map it to a `DoubleStream`, then invoke `DoubleStream` method `sum` to total the `Employee`s' salaries. Lines 118–120 also sum the `Employee`s' salaries but do so using `DoubleStream` method `reduce` rather than `sum`—the lambda in line 120 could be replaced with the `static` method reference

```
Double::sum
```

Class `Double`'s `sum` method receives two `double`s and returns their sum.

Finally, lines 124–127 calculate the `Employee`s' average salaries using `DoubleStream` method `average`, which returns an `OptionalDouble` in case the `DoubleStream` does not contain any elements. Here, we know the stream has elements, so we simply call `Option-alDouble` method `getAsDouble` to get the result.

Generative AI

1 Prompt genAIs to explain the performance benefits of short-circuit terminal operations in a Java stream pipeline.
2 Prompt genAIs to modify lines 86–96 of Fig. 14.9 to sort the departments in increasing order and sort the `Employee`s within each department by last name then first name.

14.13 Creating a `Stream<String>` from a File

Figure 14.10 uses lambdas and streams to summarize the number of occurrences of each word in a file, then display that summary in alphabetical order grouped by starting letter. Such summaries are commonly used in the natural language processing (NLP) task of similarity detection among the works of different authors. We split the program into parts for discussion purposes. Line 14 of Fig. 14.10 creates a regular expression `Pattern` that we'll use to split lines of text into their individual words. The `Pattern \s+` represents one or more consecutive white-space characters—recall that because `\` indicates an escape sequence in a `String`, we must specify each `\` in a regular expression as `\\`. As written, this program assumes that the file it reads contains no punctuation, but you could use Section 7.8's regular-expression techniques to remove punctuation.

Click here to view code image

```
1  // Fig. 14.10: StreamOfLines.java
2  // Counting word occurrences in a text file.
3  import java.io.IOException;
4  import java.nio.file.Files;
5  import java.nio.file.Paths;
6  import java.util.Map;
7  import java.util.TreeMap;
8  import java.util.regex.Pattern;
9  import java.util.stream.Collectors;
10
11 public class StreamOfLines {
12    public static void main(String[] args) throws IOException {
13       // Regex that matches one or more consecutive whitespace characters
```

```
14          Pattern pattern = Pattern.compile("\\s+");
15
```

Fig. 14.10 | Counting word occurrences in a text file.

Summarizing the Occurrences of Each Word in the File

The stream pipeline in lines 17–21 summarizes the contents of the text file `"Chapter2Paragraph.txt"` (which is located in the folder with the example) into a `Map<String, Long>` in which each `String` key is a word in the file, and the corresponding `Long` value is the number of occurrences of that word.

Click here to view code image

```
16          // count occurrences of each word in a Stream<String> sorted by word
17          Map<String, Long> wordCounts =
18             Files.lines(Paths.get("Chapter2Paragraph.txt"))
19                  .flatMap(line -> pattern.splitAsStream(line))
20                  .collect(Collectors.groupingBy(String::toLowerCase,
21                     TreeMap::new, Collectors.counting()));
22
```

The pipeline performs the following tasks:

- Line 18 calls **`Files` method `lines`** to get a `Stream<String>` that reads lines from a file and returns each as a `String`. Class `Files` (package `java.nio.file`) is one of many classes throughout the Java APIs containing methods that return `Stream`s.
- Line 19 uses **`Stream` method `flatMap`** to break each line of text into its separate words. Method `flatMap` receives a `Function` that maps an object into a stream of elements. In this case, the object is a `String` containing words, and the result is a `Stream<String>` for the individual words. The lambda in line 19 passes the `String` representing a line of text to **`Pattern` method `splitAsStream`**, which uses the regular expression specified in the `Pattern` (line 14) to tokenize the `String` into its individual words. The result of line 19 is a `Stream<String>` for the individual words in all the lines of text. (This lambda could be replaced with the method reference `pattern::splitAsStream`.)
- Lines 20–21 use `Stream` method `collect` to count the

frequency of each word and place the words and their counts into a `TreeMap<String, Long>`—a `TreeMap` maintains its keys in sorted order. Here, we use a version of `Collectors` method `groupingBy` that receives three arguments—a classifier, a `Map` factory and a downstream `Collector`. The classifier is a `Function` that returns objects for use as keys in the resulting `Map`—the method reference `String::toLowerCase` converts each word to lowercase. The `Map` factory is an object that implements the `Supplier` interface and returns a new `Map` collection. Here, we use the **constructor reference** `TreeMap::new`. The compiler converts a constructor reference into a parameterless lambda that returns a new `TreeMap`. The downstream `Collector` in this example is `Collectors.counting()`, which determines the number of occur-rences of each key in the stream. The `TreeMap`'s key type is determined by the classifier `Function`'s return type (`String`), and the value type is determined by the downstream collector—`Collectors.counting()` returns a `Long`.

Displaying the Summary Grouped by Starting Letter

Next, the stream pipeline in lines 24–33 groups the key–value pairs in the `Map wordCounts` by the keys' first letter. This produces a new `Map` in which each key is a `Character`, and the corresponding value is a `List` of the key–value pairs in `wordCounts` in which the key starts with the `Character`.

Click here to view code image

```
23          // display the words grouped by starting letter
24          wordCounts.entrySet()
25             .stream()
26             .collect(
27                Collectors.groupingBy(entry -> entry.getKey().charAt(0),
28                   TreeMap::new, Collectors.toList()))
29             .forEach((letter, wordList) -> {
30                System.out.printf("%n%C%n", letter);
31                wordList.stream().forEach(word -> System.out.printf(
32                   "%13s: %d%n", word.getKey(), word.getValue()));
33             });
34       }
35    }
```

The statement performs the following tasks:

- First, we need a `Stream` to process `wordCounts`' key–value pairs. Interface `Map` does not have methods that return `Stream`s. So, line 24 calls `Map`'s `entrySet` method to get a `Set` of **`Map.Entry` objects**, each containing a key–value pair from `wordCounts`. This produces an object of type `Set<Map.Entry<String, Long>>`.
- Line 25 calls `Set` method `stream` to get a `Stream<Map.Entry<String, Long>>`.
- Lines 26–28 call `Stream` method `collect` with three arguments—a classifier, a `Map` factory and a downstream `Collector`. This classifier gets the key from the `Map.Entry` then get the key's first character, which becomes a `Character` key in the resulting `Map`. The constructor reference `TreeMap::new` is the `Map` factory that creates a `TreeMap` that maintains the keys in sorted order. The downstream `Collector` (`Collectors.toList()`) places the `Map.Entry` objects into a `List` collection. The result of `collect` is a `Map<Character, List<Map.Entry<String, Long>>>`.
- Finally, summarize the words and their counts by letter (i.e., the concordance), lines 29–33 call `Map` method `forEach`. Its lambda (a `BiConsumer`) receives two parameters—`letter` and `wordList`. In the `Map` produced by `collect`, these represent each key–value pair's `Character` and `List`. The lambda's body has two statements, so it's enclosed in braces. Line 30 displays the `Character` on its own line. Lines 31–32 get a `Stream<Map.Entry<String, Long>>` from the `wordList`, then call `Stream` method `forEach` to display the key and value from each `Map.Entry`.

Figure 14.11 contains the program's output.

```
A
            a: 2
          and: 3
  application: 2
   arithmetic: 1

B
        begin: 1

C
   calculates: 1
 calculations: 1
      chapter: 1
```

```
     chapters: 1
  commandline: 1
     compares: 1
   comparison: 1
      compile: 1
     computer: 1
D
    decisions: 1
 demonstrates: 1
      display: 1
     displays: 2

E
      example: 1
     examples: 1

F
          for: 1
         from: 1

H
          how: 2
I
        inputs: 1
      instruct: 1
    introduces: 1

J
          java: 1
           jdk: 1

L
          last: 1
         later: 1
         learn: 1

M
          make: 1
      messages: 2
N
       numbers: 2

O
       obtains: 1
            of: 1
            on: 1
        output: 1

P
       perform: 1
```

```
       present: 1
       program: 1
   programming: 1
      programs: 2

R
        result: 1
       results: 2
           run: 1

S
          save: 1
        screen: 1
          show: 1
           sum: 1
T
          that: 3
           the: 7
         their: 2
          then: 2
          this: 2
            to: 4
         tools: 1
           two: 2

U
           use: 2
          user: 1

W
            we: 2
          with: 1

Y
        you'll: 2
```

Fig. 14.11 | Output of Fig. 14.10 arranged in four columns.

Generative AI

1 Prompt genAIs to suggest uses of Java stream pipelines for processing lines of text from a text file.

14.14 Streams of Random Values

Figure 5.3 summarized 60,000,000 rolls of a six-sided die using external iteration (a `for` loop) and a `switch` statement that determined which counter to increment. We then displayed the results using separate statements. In Fig. 6.6, we reimplemented Fig. 5.3, replacing the entire `switch` statement with a single statement that incremented counters in an array—that version of rolling the die still used external iteration to produce and summarize 60,000,000 random rolls and to display the final results. Both prior versions used mutable variables to control the external iteration and to summarize the results. Figure 14.12 reimplements those programs with a single statement that does it all, using lambdas, streams, internal iteration and no mutable variables to roll the die 60,000,000 times, calculate the frequencies and display the results.

Click here to view code image

```
1  // Fig. 14.12: RandomIntStream.java
2  // Rolling a die 60,000,000 times with streams
3  import java.util.random.RandomGenerator;
4  import java.util.function.Function;
5  import java.util.stream.Collectors;
6
7  public class RandomIntStream {
8     public static void main(String[] args) {
9        var random = RandomGenerator.getDefault();
10
11       // roll a die 60,000,000 times and summarize the results
12       System.out.printf("%-6s%s%n", "Face", "Frequency");
13       random.ints(60_000_000, 1, 7)
14             .boxed()
15             .collect(Collectors.groupingBy(Function.identity(),
16                Collectors.counting()))
17             .forEach((face, frequency) ->
18                System.out.printf("%-6d%d%n", face, frequency));
19    }
20 }
```

```
Face   Frequency
1      9992993
2      10000363
3      10002272
4      10003810
5      10000321
6      10000241
```

Fig. 14.12 | Rolling a die 60,000,000 times with streams.

`RandomGenerator` has overloaded methods `ints`, **`longs`** and **`doubles`** that return an `IntStream`, a `LongStream` or a `DoubleStream`, representing streams of random numbers. Each method has four overloads. We describe the `ints` overloads here—methods `longs` and `doubles` perform the same tasks for streams of `long` and `double` values, respectively:

- `ints()`—creates an `IntStream` for an infinite stream (Section 14.15) of random `int` values.
- `ints(long)`—creates an `IntStream` with the specified number of random `int`s.
- `ints(int, int)`—creates an `IntStream` for an infinite stream of random `int` values in the half-open range starting with the first argument and up to, but not including, the second argument.
- `ints(long, int, int)`—creates an `IntStream` with the specified number of random `int` values in the range starting with the first argument and up to, but not including, the second argument.

Line 13 uses the last overloaded version of `ints` (introduced in Section 14.6) to create an `IntStream` of 60,000,000 random integer values in the range 1–6.

Converting an `IntStream` to a `Stream<Integer>`

We summarize the roll frequencies by collecting them into a `Map<Integer, Long>` in which each `Integer` key is a side of the die, and each `Long` value is the frequency of that side. To summarize the results in a `Map`, we must first convert the `IntStream` to a `Stream<Integer>`. We do this by calling **`IntStream` method `boxed`**.

Summarizing the Die Frequencies

Lines 15–16 call `Stream` method `collect` to summarize the results into a `Map<Integer, Long>`. The first argument to `Collectors` method `groupingBy` (line 15) calls the **`Function` interface's `static` method `identity`** to create a `Function` that simply returns its argument. This allows the actual random values to be used as the `Map`'s keys. The second argument to method `groupingBy` counts the number of occurrences of each key.

Displaying the Results

Lines 17–18 call the resulting `Map`'s `forEach` method to display the summary of the results. This method receives an object that implements the `BiConsumer` functional interface as an argument. Recall that for `Map`s, the first parameter represents the key, and the second represents the corresponding value. The lambda in lines 17–18 uses parameter `face` as the key and `frequency` as the value and displays the face and frequency.

Generative AI

1 Prompt genAIs with the code in Fig. 14.12, asking them to rewrite it to avoid boxing. Ask them to compare the performance of each approach.

14.15 Infinite Streams

A data structure, such as an array or a collection, always represents a finite number of elements—all the elements are stored in memory, and memory is finite. Of course, any stream created from a finite data structure will have a finite number of elements, as has been the case in this chapter's prior examples.

Lazy evaluation makes it possible to work with **infinite streams** representing an unknown, potentially infinite number of elements. For example, you could define a method `nextPrime` that produces the next prime number in sequence every time you call it. You could then use this to define an infinite stream that conceptually represents all prime numbers. However, because streams are lazy until you perform a terminal operation, you can use intermediate operations to restrict the total number of elements calculated when a terminal operation is performed. Consider the following pseudocode stream pipeline:

Click here to view code image

```
Create an infinite stream representing all prime numbers
    If the prime number is less than 10,000
        Display the prime number
```

Though we begin with an infinite stream, only the finite set of primes less than 10,000 would be displayed. You create infinite streams with stream-interface methods `iterate` and `generate`. For this discussion, we'll use `IntStream`'s versions of these methods.

`IntStream` Method `iterate`

Consider the following infinite stream pipeline:

Click here to view code image

```
IntStream.iterate(1, x -> x + 1)
         .forEach(System.out::println);
```

`IntStream` method `iterate` generates an ordered sequence of values starting with the seed value (`1`) in its first argument. Each subsequent element is produced by applying to the preceding value in the sequence the `IntUnaryOperator` specified as `iterate`'s second argument. The preceding pipeline generates the infinite sequence 1, 2, 3, 4, 5, …, but this pipeline has a problem. We did not specify how many elements to produce, so this is the equivalent of an infinite loop.

Limiting an Infinite Stream's Number of Elements

One way to limit the total number of elements an infinite stream produces is the **short-circuiting terminal operation `limit`**, which specifies the maximum number of elements to process from a stream. In the case of an infinite stream, `limit` terminates the infinite generation of elements. So, the following stream pipeline

Click here to view code image

```
IntStream.iterate(1, x -> x + 1)
         .limit(10)
         .forEach(System.out::println);
```

begins with an infinite stream but limits the total number of elements produced to `10`, so it displays the numbers from 1 through 10. Similarly, the pipeline

Click here to view code image

```
IntStream.iterate(1, x -> x + 1)
         .map(x -> x * x)
         .limit(10)
         .sum()
```

starts with an infinite stream but sums only the squares of the integers from 1 through 10. Ensure that stream pipelines using methods that produce infinite streams `limit` the number of elements to produce.

`IntStream` Method `generate`

You also may create unordered infinite streams using **`IntStream` method `generate`**, which receives an `IntSupplier` representing a method that takes no arguments and returns an `int`. For example, if you have a `RandomGenerator` object named `random`, the following stream pipeline generates and displays 10 random integers:

Click here to view code image

```
IntStream.generate(() -> random.nextInt())
         .limit(10)
         .forEach(System.out::println);
```

This is equivalent to using `RandomGenerator`'s no-argument `ints` method (Section 14.14):

Click here to view code image

```
random.ints()
      .limit(10)
      .forEach(System.out::println);
```

Generative AI

1 Prompt genAIs for common uses of infinite streams.

14.16 Additional Notes on Interfaces

Interfaces Allow Inheritance of Method Implementations

When a class implements an interface with `default` methods and does not override them, the class inherits the `default` methods' implementations. Recall that one class can implement many interfaces. If a class implements two or more unrelated interfaces

that provide a `default` method with the same signature, the implementing class must override that method; otherwise, a compilation error occurs.

`@FunctionalInterface` Annotation

You can create custom functional interfaces by ensuring each contains one `abstract` method and zero or more `default` and `static` methods. Though not required, you can precede a functional interface with the **`@FunctionalInterface` annotation**. The compiler will then ensure that the interface contains only one `abstract` method; otherwise, it will generate a compilation error.

14.17 Wrap-Up

In this chapter, you worked with lambdas, streams and functional interfaces. We presented many examples, often demonstrating simpler ways to implement tasks you programmed in earlier chapters.

You processed elements in an `IntStream`—a stream of `int` values. You created an `IntStream` representing a closed range of `int`s, then used intermediate and terminal stream operations to create and process a stream pipeline that produced a result. We used lambdas to create anonymous methods that implemented functional interfaces and passed lambdas to methods in stream pipelines to specify the processing steps for the streams' elements. We also created `IntStream`s from existing arrays of `int` values.

We discussed how a stream's intermediate processing steps are applied to each element before moving to the next. We used a `forEach` terminal operation to perform an operation on each stream element. We also used reduction operations to count stream elements, determine the minimum and maximum values, and sum and average the values. You also used method `reduce` to create your own reduction operations.

You used intermediate operations to filter elements that matched a predicate and map elements to new values. In each case, these operations produced intermediate streams on which you could perform additional processing. You also learned how to sort elements in ascending and descending order and how to sort objects by multiple fields.

We demonstrated how to store a stream pipeline's results in a collection using various predefined `Collector` implementations

provided by class `Collectors`. You also learned how to use a `Collector` to group elements into categories.

You learned various stream data sources. For example, you used `Files` method `lines` to get a `Stream<String>` that read lines from a file and used `RandomGenerator` method `ints` to get an `IntStream` of random values. You also learned how to convert an `IntStream` into a `Stream<Integer>` so that you could use `Stream` method `collect` to summarize the frequencies of the `Integer` values and store the results in a `Map`.

We introduced infinite streams and demonstrated how to limit the number of elements they generate. Finally, we presented some additional information about interfaces and streams. In the next chapter, we begin our three-chapter introduction to graphical user interfaces (GUIs), graphics and multimedia using JavaFX. In Chapter 18, Concurrency: Platform Threads to Virtual Threads, we'll parallelize stream pipelines.

Special Section: Generative AI Exercises

For each of the following questions, try multiple genAIs to determine which performs best. Always verify the results. Keep in mind that they sometimes make mistakes and even hallucinate, creating unreliable information.

14.1 ***(Refactoring Collection Code)*** Prompt genAIs with Fig. 12.1's code, asking them to refactor it using Java lambdas and streams techniques.

14.2 ***(Refactoring Collection Code)*** Prompt genAIs with Fig. 12.2's code, asking them to refactor it using Java lambdas and streams techniques.

14.3 ***(Refactoring Collection Code)*** Prompt genAIs with Fig. 12.11's code, asking them to refactor it using Java lambdas and streams techniques.

14.4 ***(Immutability)*** Prompt genAIs for an explanation of the benefits of immutability in the context of Java lambdas and streams.

14.5 ***(Optional Objects)*** Prompt genAIs for a tutorial on the various ways to use `Optional` objects, including examples of using `Optional` methods `isPresent` vs. `ifPresent`.

14.6 ***(Exception Handling with Streams)*** Prompt genAIs to explain

and provide examples of how you can use `Optional` objects to deal with exceptions encountered during processing of a Java stream pipeline.

14.7 ***(Stream Processing of Stack Traces with StackWalker)*** Prompt genAIs for a tutorial on using Java's `StackWalker` class for stream processing of stack traces.

14.8 ***(Loop vs. Stream Performance)*** Prompt genAIs to discuss the performance of traditional loops vs. Java stream pipelines in terms of memory and computational overhead.

14.9 ***(Streams Best Practices)*** Prompt GenAIs for a list of best practices when using Java lambdas and streams.

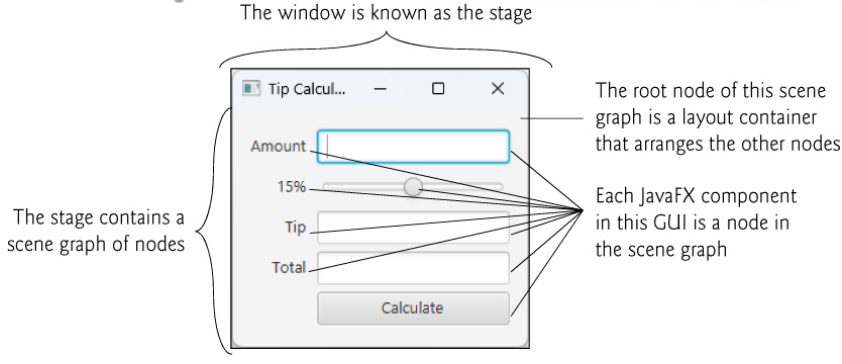
14.10 ***(Streams Pitfalls)*** Prompt GenAIs for a list of common pitfalls when using Java lambdas and streams.

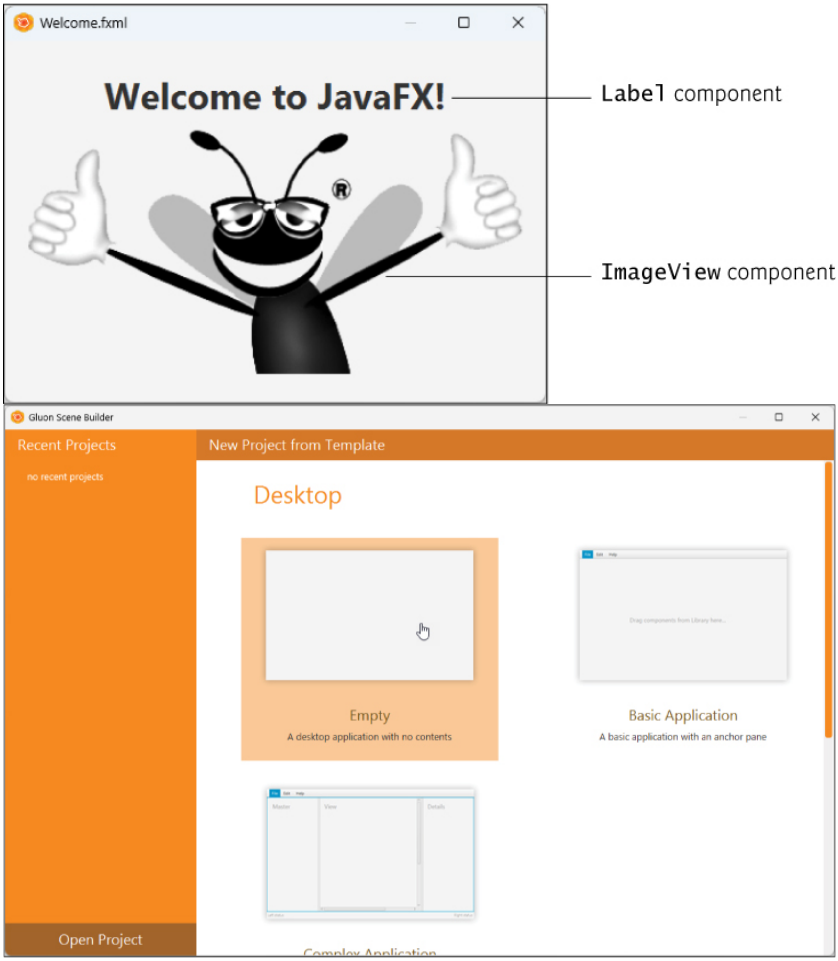
14.11 ***(Collectors)*** Prompt genAIs for code examples demonstrating the most commonly used collectors in Java's `Collectors` class.

14.12 ***(switch Expressions in Stream Pipelines)*** Prompt genAIs for examples of using `switch` expressions when mapping elements in a stream pipeline.

14.13 ***(Stream Method mapMulti)*** Prompt genAIs for examples of using the Java Stream interface's `mapMulti` method to map each element to multiple elements.

14.14 ***(Testing and Debugging Stream Pipelines)*** Prompt genAIs for guidelines on testing and debugging code stream pipelines.

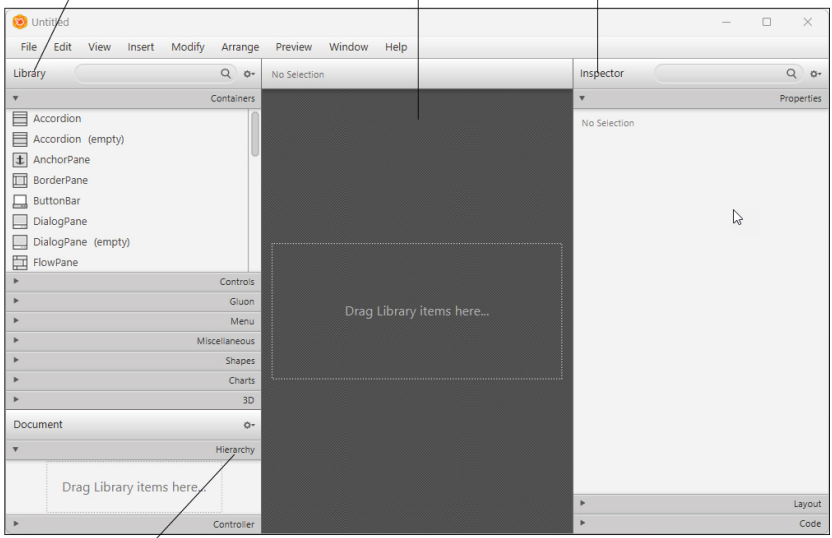

The window is known as the stage
Tip Calcul...
Amount
15%
Tip
Total
Calculate
The stage contains a scene graph of nodes
The root node of this scene graph is a layout container that arranges the other nodes
Each JavaFX component in this GUI is a node in the scene graph


Welcome.fxml
Welcome to JavaFX!
Label component
ImageView component
Gluon Scene Builder
Recent Projects
no recent projects
New Project from Template
Desktop
Empty
A desktop application with no contents
Basic Application
A basic application with an anchor pane
Open Project

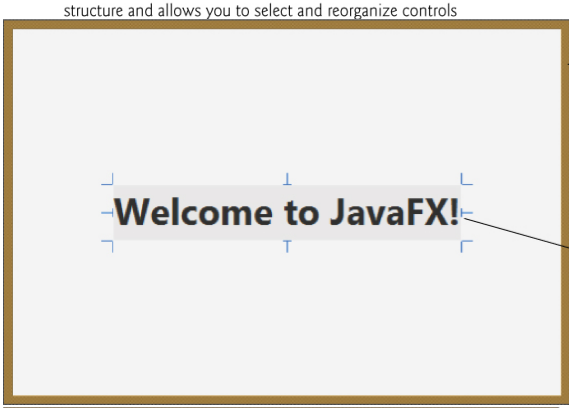
The **Library** contains JavaFX **Containers**, **Controls** and other items that can be dragged and dropped on the canvas

You use the content panel to design the GUI

You use the **Inspector** window to configure the currently selected item in the content panel

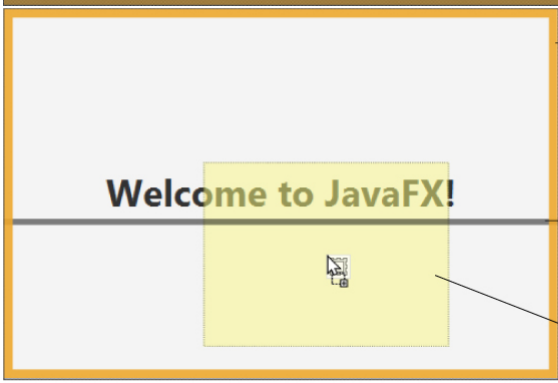


The **Document** window's **Hierarchy** section contains the GUI's structure and allows you to select and reorganize controls



Scene Builder highlights the container in which the selected control appears

Selected control—you can drag the blue handles at the sides and corners to resize the control



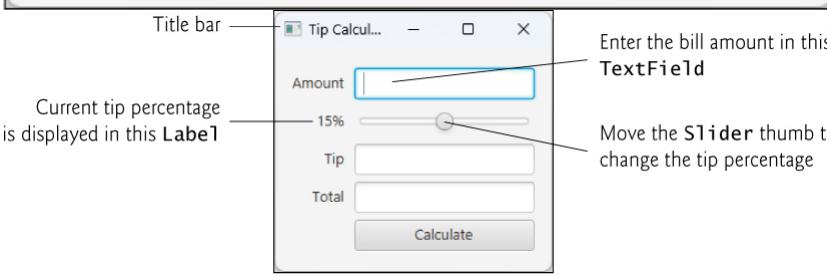
Scene Builder highlights the container in which the control will be placed

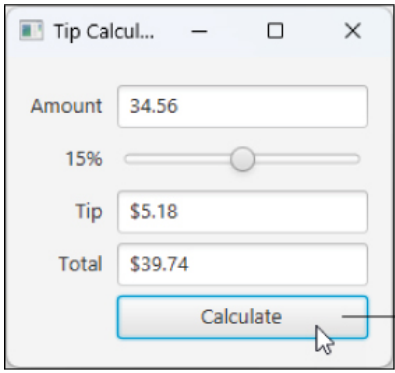
For a **VBox**, Scene Builder indicates where the new control will be placed by displaying a gray guide line

Scene Builder displays a yellow box representing the control you're dropping onto the design

Welcome to JavaFX!

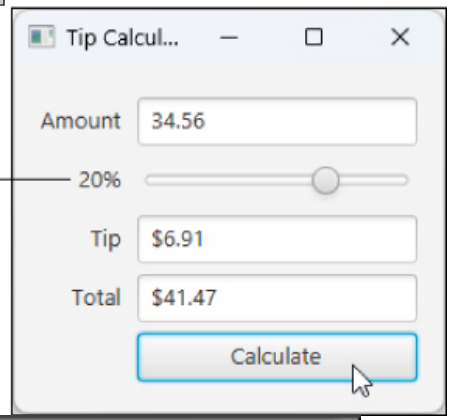
Welcome.fxml
Welcome to JavaFX!

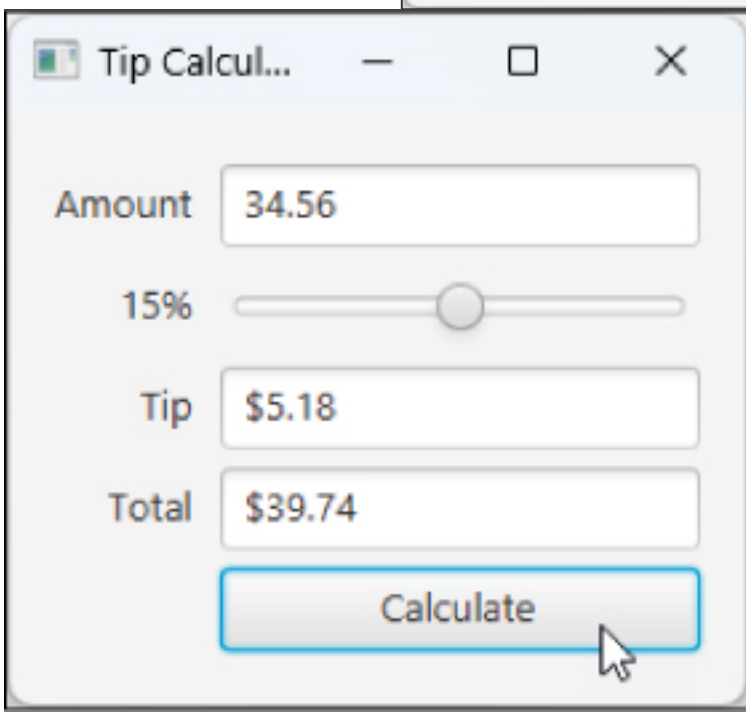

Title bar
Tip Calcul...
Amount
15%
Tip
Total
Calculate
Enter the bill amount in this TextField
Current tip percentage is displayed in this Label
Move the Slider thumb to change the tip percentage

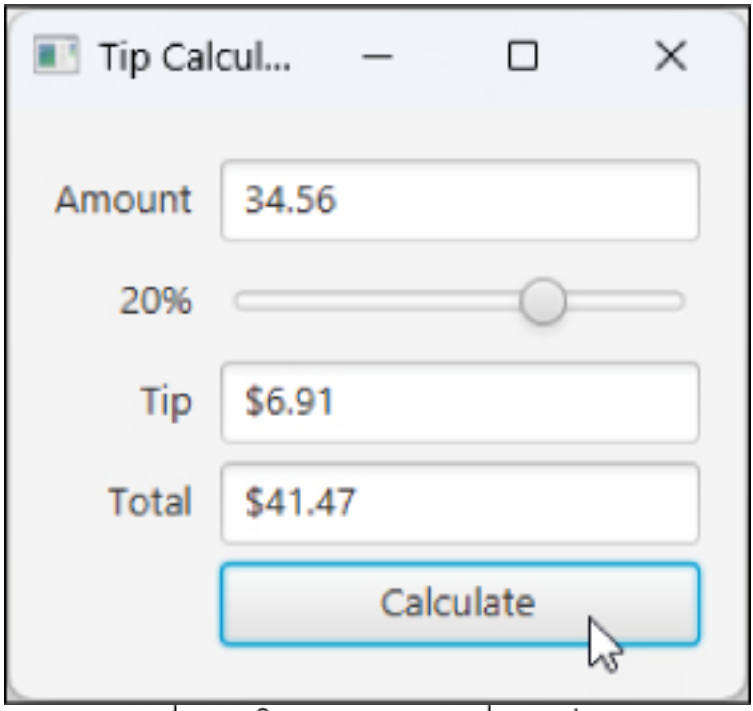


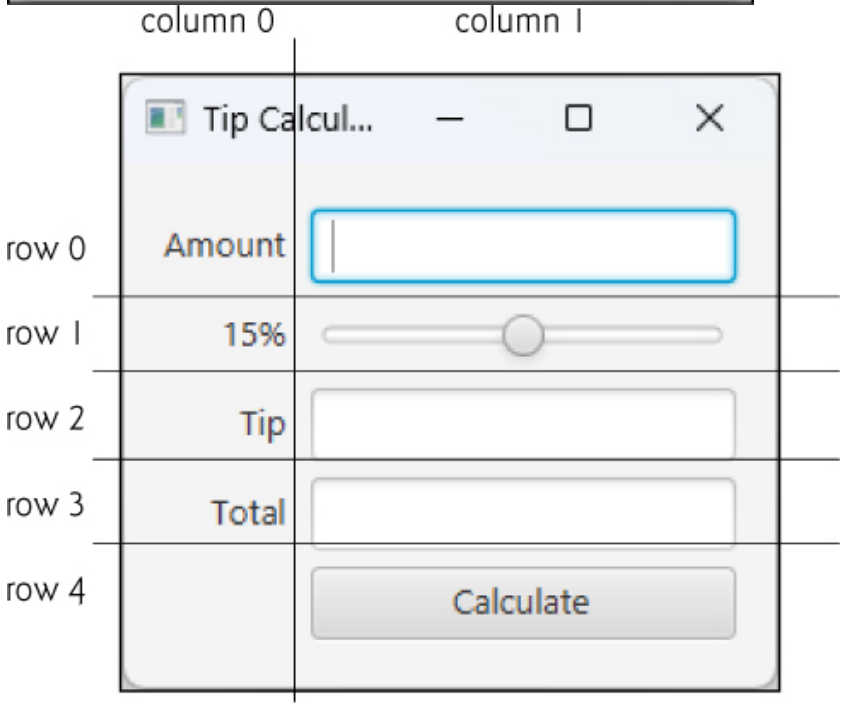
Click the **Calculate** `Button` to display the tip and total

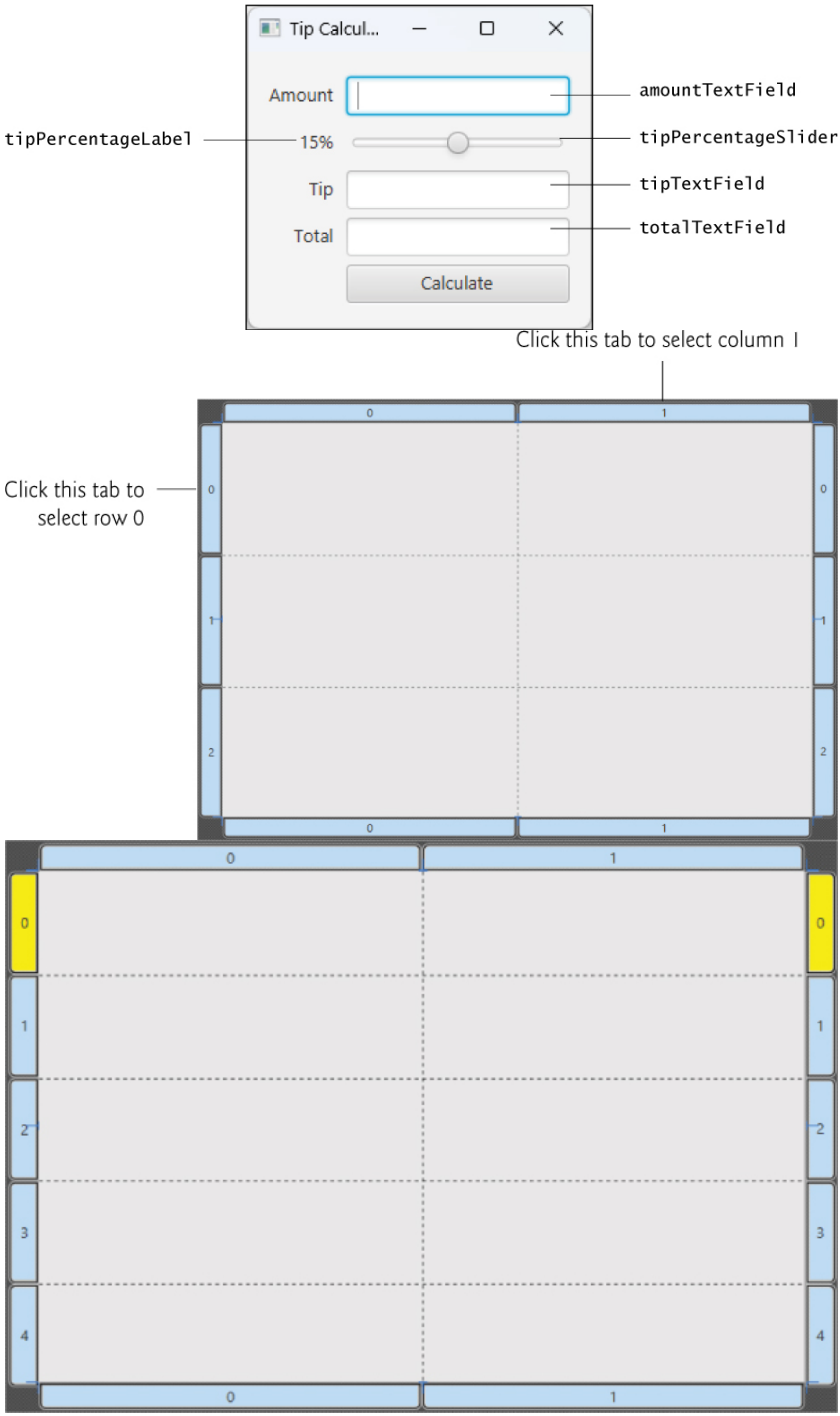
Updated tip percentage after the user moved the `Slider`'s thumb

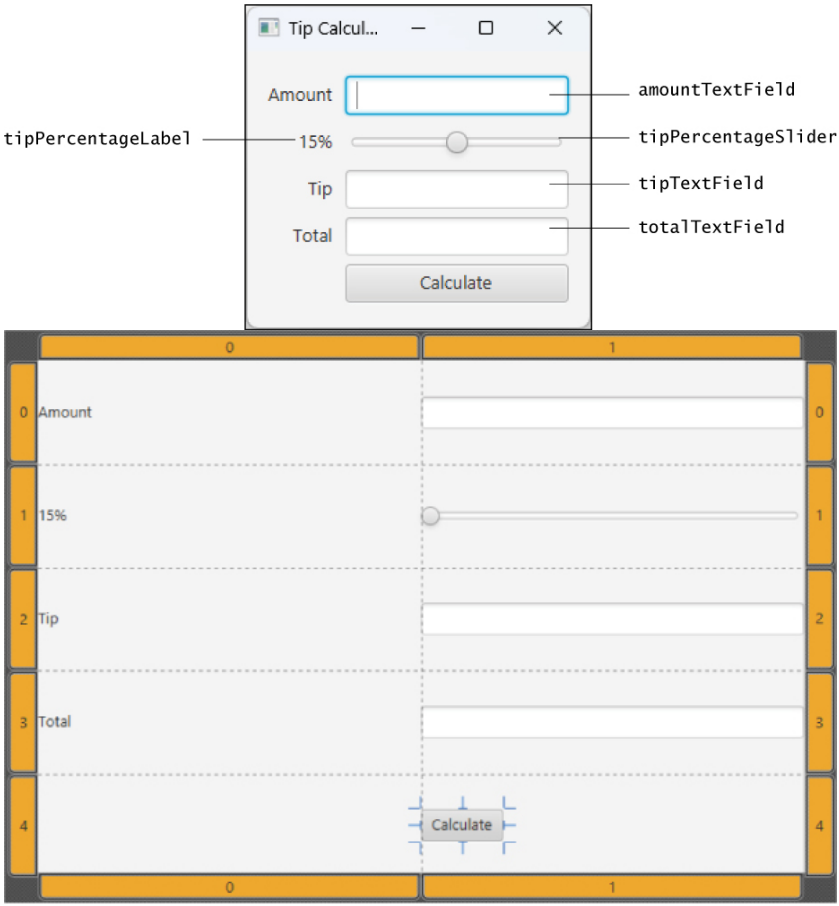


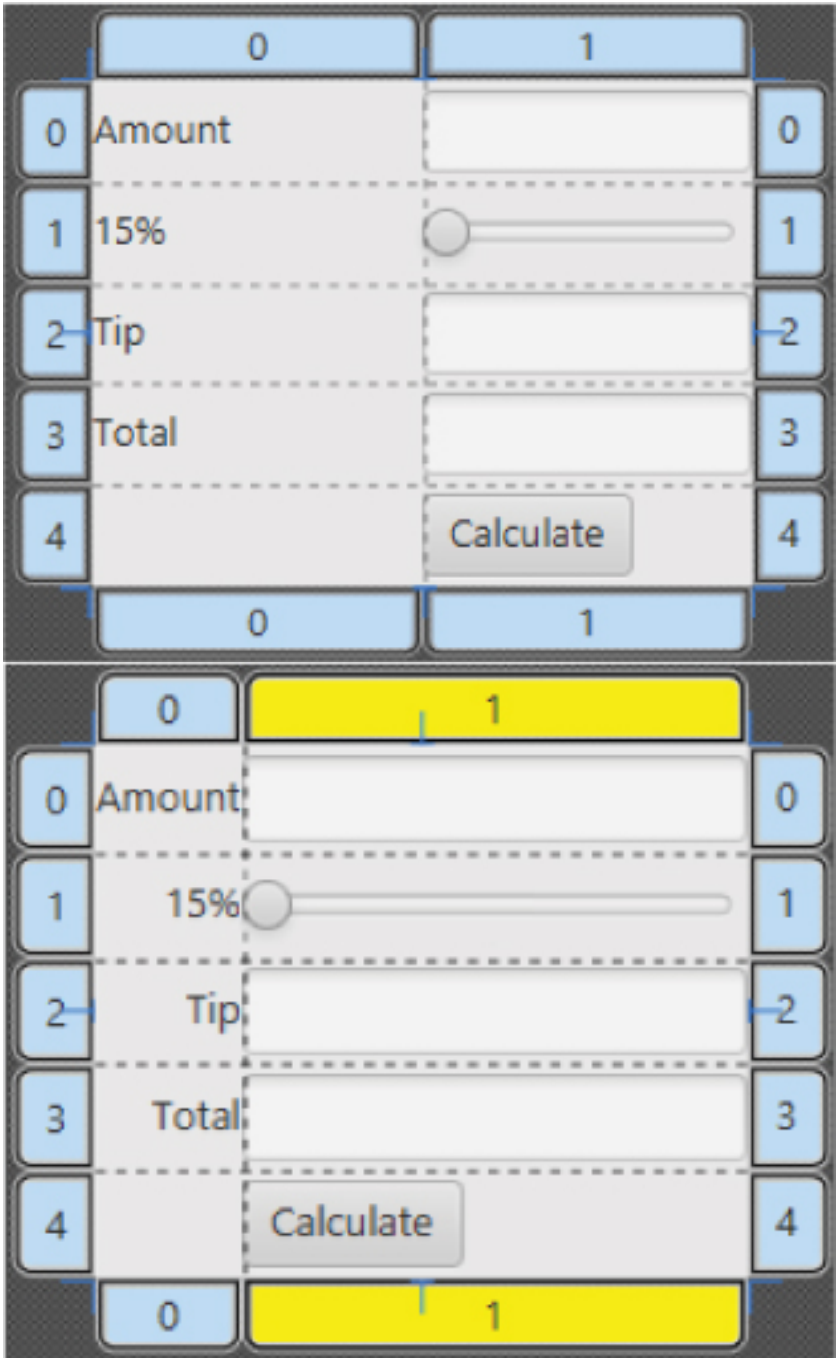


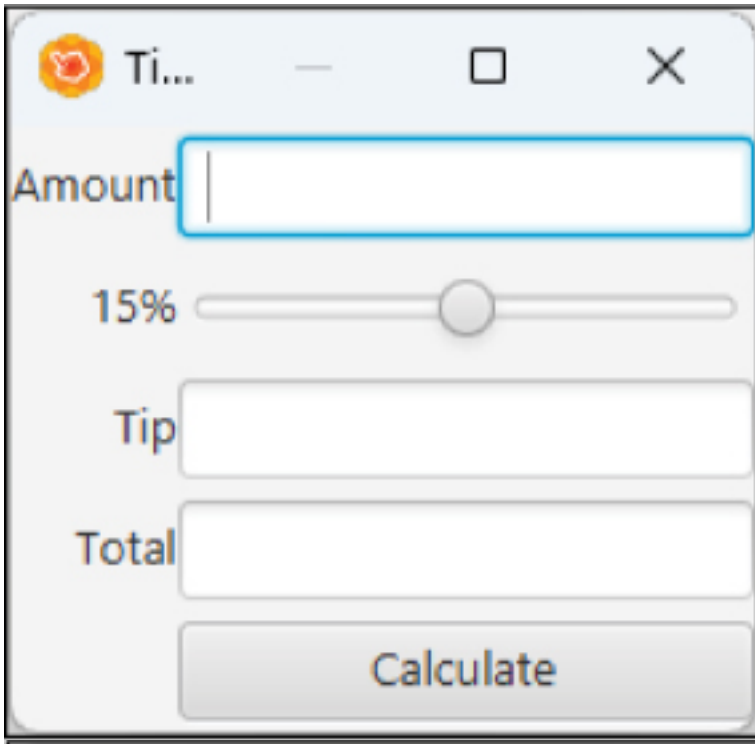

Tip Calcul...
Amount
34.56
20%
Tip
$6.91
Total
$41.47
Calculate

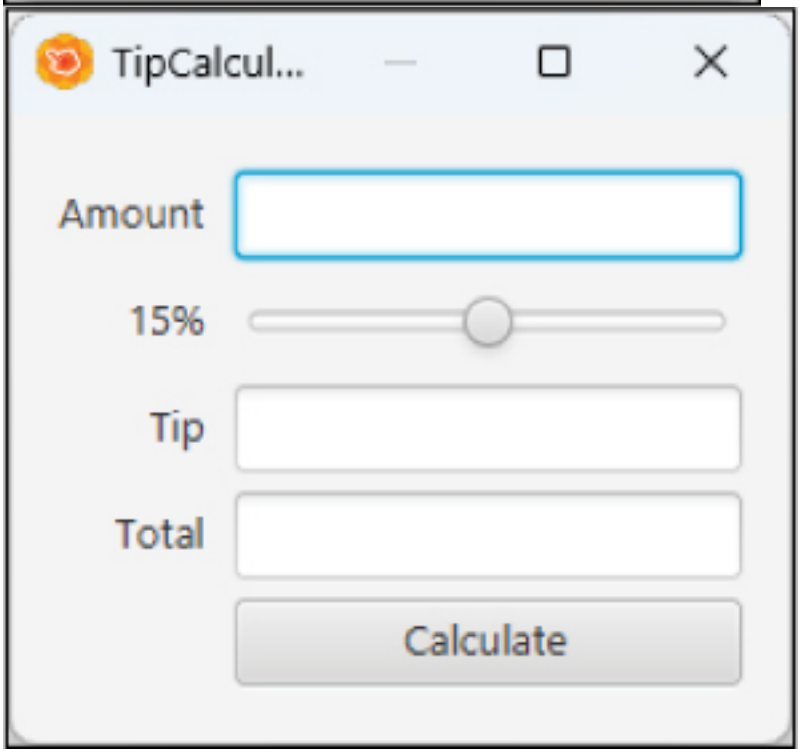

column 0
column 1
Tip Calcul...
row 0
Amount
row 1
15%
row 2
Tip
row 3
Total
row 4
Calculate

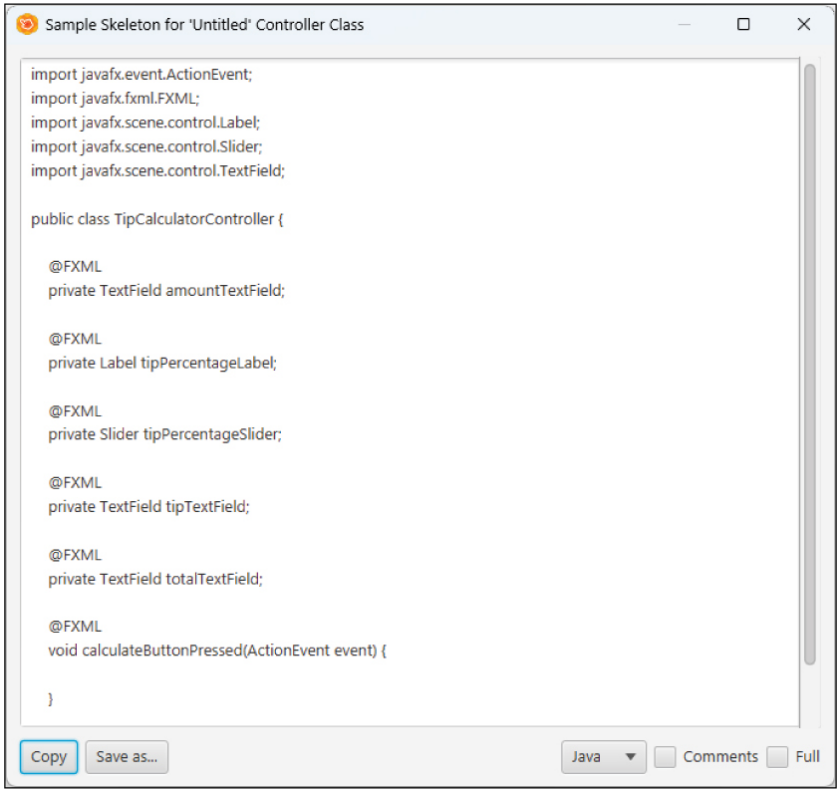

Tip Calcul...
Amount
15%
Tip
Total
Calculate
amountTextField
tipPercentageSlider
tipTextField
totalTextField
tipPercentageLabel
Click this tab to select column 1
Click this tab to select row 0


Tip Calcul...
Amount
amountTextField
tipPercentageLabel
15%
tipPercentageSlider
Tip
tipTextField
Total
totalTextField
Calculate
0
1
0
Amount
1
15%
2
Tip
3
Total
4
Calculate


0
1
0
Amount
0
1
15%
1
2
Tip
2
3
Total
3
4
Calculate
4
0
1
0
1
0
Amount
0
1
15%
1
2
Tip
2
3
Total
3
4
Calculate
4
0
1


Ti...
Amount
15%
Tip
Total
Calculate


TipCalcul...
Amount
15%
Tip
Total
Calculate



JavaFX Graphical User Interfaces: Part 1

Objectives

In this chapter you'll:

- Build JavaFX GUIs and handle events generated by user interactions with them.
- Understand the structure of a JavaFX application window.
- Use JavaFX Scene Builder to create FXML files that describe JavaFX scenes containing `Label`s, `ImageView`s, `TextField`s, `Slider`s and `Button`s without writing any code.
- Arrange GUI components using the `VBox` and `GridPane` layout containers.
- Use a controller class to define event handlers for a JavaFX FXML GUI.
- Define an event handler using a lambda.
- Work step-by-step through building a JavaFX Welcome application with a simple GUI layout.
- Work step-by-step through building a JavaFX Tip Calculator application with a more complex GUI layout, more controls and event handling to respond to user interactions.

Outline

15.1 Introduction

15.2 JavaFX Scene Builder

15.3 JavaFX Application Window Structure

15.4 Welcome GUI: Displaying Text and an Image

15.4.1 Opening Scene Builder and Creating the File `Welcome.fxml`

15.4.2 Adding an Image to the Folder Containing `Welcome.fxml`

15.4.3 Creating a `VBox` Layout Container

15.4.4 Configuring the `VBox` Layout Container

15.4.5 Adding and Configuring a `Label`

15.4.6 Adding and Configuring an `ImageView`

15.4.7 Previewing the `Welcome` GUI

15.5 Tip Calculator Application: Intro to Event Handling

15.5.1 Test-Driving the Tip Calculator

15.5.2 Technologies Overview
15.5.3 Building the Application's GUI
15.5.4 `TipCalculator` Class
15.5.5 `TipCalculatorController` Class; Implementing an Event Handler with a Lambda
15.6 Features Covered in the Other JavaFX Chapters
15.7 Wrap-Up
Special Section: Generative AI Exercises

15.1 Introduction

[Chapters 15–17 assume you've already installed the JavaFX Software Development Kit (SDK) and Scene Builder as discussed in this book's Before You Begin section.]
A graphical user interface (GUI) presents a user-friendly mechanism for interacting with an application. A GUI (pronounced "GOO-ee") gives an application a distinctive "look-and-feel." GUIs are built from **GUI components**—also called controls or widgets (short for window gadgets). A GUI component is an object with which the user interacts via the mouse, the keyboard or another form of input, such as voice recognition. Providing different apps with consistent, intuitive user-interface components gives users a sense of familiarity with a new application so that they can learn it more quickly and use it more productively.

History of GUI in Java

Java's original GUI library was the Abstract Window Toolkit (AWT). Swing was Java's primary GUI technology until JavaFX. Swing will remain part of Java and is still widely used.

JavaFX is the "next generation client application platform for desktop, mobile and embedded systems built on Java."[1] It provides GUI, graphics and multimedia APIs, as well as extensive customization capabilities. Oracle open-sourced JavaFX in 2011. With 2018's Java 11 release, JavaFX became a separate open-source project that evolves in parallel with the Java Development Kit and must be installed separately (discussed in the book's Before You

Begin section). Some of the benefits of JavaFX over Swing include:

1. “JavaFX.” Accessed September 7, 2024. https://openjfx.io/.

- JavaFX is easier to use—it provides one API for client functionality, including GUI, graphics and multimedia (images, animation, audio and video). Swing is only for GUIs, so you need to use other APIs for graphics and multimedia apps.
- With Swing, many IDEs provided GUI design tools for dragging and dropping components onto a layout; however, each IDE produced different code (such as different variable and method names). JavaFX Scene Builder (Section 15.2) can be used standalone or integrated with many IDEs, and it produces the same code regardless of the IDE.
- Though Swing components could be customized, JavaFX gives you complete control over a JavaFX GUI’s look-and-feel via **Cascading Style Sheets (CSS)**—the same technology used to style web pages. You’ll use JavaFX CSS in Chapter 17. JavaFX 23 includes support for CSS-based animations (Section 17.10).
- JavaFX has better threading support, which is important for getting the best program performance on today’s multi-core systems.
- JavaFX uses the GPU (graphics processing unit) for hardware-accelerated rendering.
- JavaFX supports transformations for repositioning and reorienting JavaFX components, and animations for changing the properties of JavaFX components over time. These can be used to make apps more intuitive and easier to use.
- JavaFX provides multiple upgrade paths for enhancing existing GUIs—Swing GUI capabilities may be embedded into JavaFX apps via class `SwingNode` and JavaFX capabilities may be embedded into Swing apps via class `JFXPanel`.
- JavaFX organizes its capabilities using the Java Platform Module System (JPMS), which we discuss in Chapter 21.

This chapter introduces JavaFX GUI basics, including layout, controls and event handlers, which we’ll implement using lambdas. We present a more detailed treatment of Java FX GUI in the next chapter. Chapter 17 discusses JavaFX graphics and multimedia.

Generative AI

1 Prompt genAIs asking why JavaFX was separated from the Java Development Kit starting with Java 11.

15.2 JavaFX Scene Builder

The **Scene Builder** tool, which is maintained by Gluon (https://gluonhq.com/), is a standalone JavaFX GUI visual layout tool. You can integrate it with various popular Java IDEs to conveniently open JavaFX GUI files directly in Scene Builder.

JavaFX Scene Builder enables you to create GUIs by dragging and dropping GUI components from Scene Builder's library onto a design area, then modifying and styling the GUI—all without writing any code. JavaFX Scene Builder's live editing and preview features allow you to view your GUI as you create and modify it—without compiling and running the application. You can use Cascading Style Sheets (CSS) to change the entire look-and-feel of your GUI—a concept sometimes called **skinning**. In Chapter 16, we'll introduce styling with CSS.

FXML (FX Markup Language)

As you create and modify a GUI, JavaFX Scene Builder generates **FXML (FX Markup Language)**—an XML vocabulary for defining and arranging JavaFX GUI controls without writing any Java code. XML (eXtensible Markup Language) is a widely used language for describing things—it's readable both by computers and by humans. In JavaFX, FXML concisely declares GUI, graphics and multimedia elements. You do not need to know FXML or XML to study this chapter. JavaFX Scene Builder hides the FXML details from you, so you can focus on defining what the GUI should contain without specifying how to generate it—this is an example of declarative programming.[2]

2. Gupta, Mala, "LiveStream Summary: JavaFX, the Cross-platform UI Development in Java." Accessed September 8/2024. https://blog.jetbrains.com/idea/2021/02/livestream-summary-javafx-the-cross-platform-ui-development-in-java/.

The FXML code is separate from the program logic that's defined in Java source code—this separation of the interface (the GUI) from

the implementation (the Java code) makes it easier to test, debug, modify and maintain JavaFX GUI apps.

Generative AI

1 Prompt genAIs for how generative AI itself is an example of declarative programming.

15.3 JavaFX Application Window Structure

A JavaFX application window consists of several parts:

Controls

Controls are GUI components, such as **`Label`**s that display text, **`TextField`**s that enable a program to receive user input, **`Button`**s that users click to initiate actions, and more.

Stage

The window in which a JavaFX application's GUI is displayed is known as the **stage** and is an instance of **class** **`Stage`** (package `javafx.stage`).

Scene

The stage contains one active **scene** that defines the GUI as a **scene graph**—a tree data structure of an application's visual elements, such as GUI controls, shapes, images, video, text and more. The scene is an instance of class **`Scene`** (package `javafx.scene`).

Nodes

Each visual element in the scene graph is a **node**—an instance of a subclass of **`Node`** (package `javafx.scene`), which defines common attributes and behaviors for all nodes. Except for the scene graph's **root node**, each node has one parent. Nodes can have transforms (e.g., moving, rotating and scaling), opacity (whether a node is transparent, translucent or opaque), effects (e.g., drop shadows,

blurs, reflection and lighting) and more that we'll introduce in Chapter 16.

Layout Containers

Nodes that have children are typically **layout containers** that arrange their child nodes in the scene. You'll use two layout containers (`VBox` and `GridPane`) in this chapter and learn several more in Chapters 16–17. The nodes arranged in a layout container are a combination of controls and, in more complex GUIs, possibly other layout containers.

Event Handler and Controller Class

When the user interacts with a control, such as clicking a `Button` or entering text into a `TextField`, the control generates an **event**. Programs can respond to these events—known as event handling—to specify what should happen when each user interaction occurs. An **event handler** is a method that responds to a user interaction. An FXML GUI's event handlers are defined in a **controller class** (Section 15.5.5).

Generative AI

1 Prompt genAIs to explain how, with GUI-based programs and event handling, the user "drives the program's actions" rather than vice versa as with non-GUI user interfaces.

15.4 Welcome Application: Displaying Text and an Image

In this section, without writing any code, you'll build a GUI that displays text in a `Label` and an image in an `ImageView`:
You'll use visual-programming techniques to drag-and-drop JavaFX components onto Scene Builder's content panel (the design area), then use the Inspector to configure options, such as the `Labels`'s text and font size and the `ImageView`'s image. You'll view the completed GUI using Scene Builder's Show Preview in Window option. In Section 15.5's Tip Calculator application, we'll discuss the Java code that loads and displays an FXML GUI.

15.4.1 Opening Scene Builder and Creating the File `Welcome.fxml`

Open Scene Builder so that you can create the FXML file that defines the GUI. Initially, the following Gluon Scene Builder window appears. In it, you can choose either

- a recent project to edit (if you have any) from the Recent Projects column, or
- one of Scene Builder's preexisting application design templates from the New Project from Template column.

Select the Empty project template to create a JavaFX GUI from scratch. After doing so, the SceneBuilder window will appear as follows:
Untitled at the top of the window indicates that Scene Builder created a new FXML file that you have not yet saved.[3] Select File > Save to display the Save As dialog, then select a location in which to store the file, name the file `Welcome.fxml` and click the Save button.

3. We present the Scene Builder screen captures using Microsoft Windows 11, but Scene Builder is nearly identical on Windows, macOS and Linux. The primary difference is that the menu bar on macOS is at the top of the screen, whereas the menu bar is part of the window on Windows and Linux.

15.4.2 Adding an Image to the Folder Containing `Welcome.fxml`

The image you'll use for this application (`bug.png`) is located in the `images` subfolder of this chapter's examples folder. To make it easy to find the image when you're ready to add it to the application, locate the `images` folder on your file system, then copy `bug.png` into the folder where you saved `Welcome.fxml`.

15.4.3 Creating a `VBox` Layout Container

For this application, you'll place a `Label` and an `ImageView` in a **`VBox` layout container** (package `javafx.scene.layout`), which will be the scene graph's root node. Layout containers help you arrange and size GUI components. A `VBox` arranges its nodes vertically from

top to bottom. We discuss the `GridPane` layout container in Section 15.5 and several others in Chapter 16. To add a `VBox` to Scene Builder's content panel so you can begin designing the GUI, double-click VBox in the Library window's Containers section. (You also can drag-and-drop a `VBox` from the Containers section onto Scene Builder's content panel.)

15.4.4 Configuring the `VBox` Layout Container

You'll now specify the `VBox`'s alignment, initial size and padding.

Specifying the `VBox`'s Alignment

A `VBox`'s **alignment** determines the layout positioning of the `VBox`'s children. In this application, we'd like each child node (the `Label` and the `ImageView`) to be centered horizontally in the scene, and we'd like both children to be centered vertically, so that there is an equal amount of space above the `Label` and below the `ImageView`. To accomplish this:

- Select the `VBox` in Scene Builder's content panel by clicking it. Scene Builder displays many `VBox` properties in the Scene Builder Inspector's Properties section.
- Click the Alignment property's drop-down list and notice the variety of potential alignment values you can use. Click `CENTER` to set the Alignment.

Each property value you specify for a JavaFX object is used to set one of that object's instance variables when JavaFX creates the object at runtime.

Specifying the `VBox`'s Preferred Size

The **preferred size** (width and height) of the scene graph's root node is used by the scene to determine its window size when the application begins executing—for this application, we arbitrarily chose 450-by-300 pixels for the `VBox`'s size, which works well for the text and image we'll display. To set the preferred size:

- Select the `VBox`.
- Expand the Inspector's Layout section by clicking the right

arrow ( ) next to Layout. The section expands, and the right arrow changes to a down arrow. Clicking the arrow again would collapse the section.

- Click the Pref Width property's text field, enter `450`, and press *Enter* to change the preferred width.
- Click the Pref Height property's text field, enter `300`, and press *Enter* to change the preferred height.

As you change these properties, the `VBox`'s size changes immediately, so you can confirm your modifications were applied.

15.4.5 Adding and Configuring a `Label`

Next, you'll create the `Label` that displays `"Welcome to JavaFX!"`.

Adding a `Label` to the `VBox`

Expand the Scene Builder Library window's Controls section by clicking the right arrow ( ) next to Controls, then drag-and-drop a Label from the Controls section onto the `VBox` in Scene Builder's content panel. (You also can double-click Label in the Containers section to add the `Label`.) Scene Builder automatically centers the `Label` object horizontally and vertically in the `VBox`, based on the `VBox`'s Alignment property.

Changing the `Label`'s Text

You can set a `Label`'s text either by double-clicking it and entering the new text or by selecting the `Label` and setting its Text property in the Inspector's Properties section. Set the `Label`'s text to `"Welcome to JavaFX!"`.

Changing the `Label`'s Font

For this application, we set the `Label` to display in a large, bold font. To do so, select the `Label`, then in the Inspector's Properties section, click the down arrow ( ) to the right of the Font property. In the window that appears, set the Style property to `Bold` and the Size property to `30`. The design should now appear as follows:

15.4.6 Adding and Configuring an `ImageView`

Finally, you'll add the `ImageView` that displays `bug.png`.

Adding an `ImageView` to the `VBox`

Drag and drop an ImageView from the Library window's Controls section to just below the `Label`, as follows—put the image below the gray guide line:

You can also double-click ImageView in the Library window, in which case Scene Builder automatically places the new `ImageView` object below the `Label`. You can reorder a `VBox`'s controls by dragging them in the `VBox` or in the Document window's Hierarchy section. Scene Builder automatically centers the `ImageView` horizontally in the `VBox`. Also, notice that the `Label` and `ImageView` are centered vertically such that the same amount of space appears above the `Label` and below the `ImageView`.

Setting the `ImageView`'s Image

Next, you'll set the image to display:

- Select the `ImageView`, then in the Inspector's Properties section, click the ellipsis (…) button to the right of the Image property. By default, Scene Builder opens a dialog listing the contents of the folder in which the FXML file is saved. This is where you placed the image file `bug.png` in Section 15.4.2.
- Select `bug.png`, then click Open. Scene Builder displays the image and resizes the `ImageView` to match the image's aspect ratio—that is, the ratio of the image's width to its height.

Changing the `ImageView`'s Size

We'd like to display the image at its original size. If you reset the `ImageView`'s default Fit Width and Fit Height property values—which Scene Builder set when you added the `ImageView` to the design—Scene Builder will resize the `ImageView` to the image's exact dimensions. To reset these properties:

- Expand the Inspector's Layout section.
- Hover the mouse over the Fit Width property's value. This displays a small settings button ( ). Click the button and select Reset to default to reset the value. This

technique can be used with any property value to reset its default.

- Repeat Step 2 to reset the Fit Height property's value.

Scene Builder's content panel with the completed GUI should now appear as follows:

Save the FXML file by selecting File > Save.

15.4.7 Previewing the `Welcome` GUI

You can preview what the design will look like in a running program's window by selecting Preview > Show Preview in Window, which displays the following window:

The window's content will be identical across Linux, macOS and Windows—only the window borders will differ. JavaFX uses the operating system's default window decorations—such as the icons for minimizing, maximizing and closing a window.

Generative AI

1 Prompt genAIs to comment on how GUI development "feels different" from programming with "conventional" non-GUI techniques.

2 Prompt genAIs asking for a list of JavaFX's accessibility features and recommendations for making your JavaFX GUIs accessible for people who are blind or have low vision.

15.5 Tip Calculator Application: Intro to Event Handling

The Tip Calculator application calculates and displays a restaurant bill tip and total:

By default, the application calculates the total with a 15% tip. The following screen capture contains the GUI after you enter the amount `34.56` and click the Calculate `Button`:

You can specify a tip percentage from 0% to 30% by moving the `Slider` thumb—this updates the tip percentage. The following screen capture contains the GUI after the user moves the `Slider`'s thumb to change the tip percentage to 20%, then clicks Calculate:

In this section, you'll build the Tip Calculator application using several JavaFX components and learn how to respond to user interactions with the GUI. You'll begin by test-driving the application, using it to calculate 15% and 20% tips. Then, we'll overview the technologies you'll use to create the application. You'll build the application's GUI using the Scene Builder. Finally, we'll present the complete Java code for the application and do a detailed code walkthrough.

15.5.1 Test-Driving the Tip Calculator

Let's compile and run the application, which is located in the `TipCalculator` folder with this chapter's examples. Class `TipCalculator` is the main application class.

Compiling the Tip Calculator Application

As we mentioned, JavaFX uses the Java Platform Module System (JPMS; Chapter 21) to organize its capabilities. To compile and run the Tip Calculator application, you must include two `javac` command-line JPMS options:

- The **`--module-path` option** specifies where the JavaFX modules are located on your system—the `PATH_TO_FX` environment variable you created in this book's Before You Begin section specifies the JavaFX modules' location.
- The **`--add-modules` option** specifies the JavaFX modules containing the capabilities required by the application. In this case, the **`javafx.controls` module** contains the JavaFX API features that define controls, such as `Label`s, `TextField`s, `Button`s and `Slider`s, and the **`javafx.fxml` module** provides the capabilities we'll use in the app to construct the GUI from the FXML file.

To compile the application, enter the following command in its entirety before pressing *Enter*:

Click here to view code image

```
javac --module-path pathToFX
   --add-modules javafx.controls,javafx.fxml *.java
```

replacing *pathToFX* with

- `$PATH_TO_FX` if you are using macOS,
- `$env:PATH_TO_FX` if you are using a PowerShell window on Microsoft Windows or
- `%PATH_TO_FX%` if you are using a Command Prompt window on Microsoft Windows.

Running the Tip Calculator Application

To run the application, enter the following command in its entirety before pressing *Enter*:

Click here to view code image

```
java --module-path pathToFX
   --add-modules javafx.controls,javafx.fxml TipCalculator
```

The same `--module-path` and `--add-modules` options are required for both the `javac` and `java` commands.

Entering a Bill Total

Use your keyboard to enter the bill amount `34.56` into the Amount `TextField`, then click the Calculate `Button`. The Tip and Total `TextField`s contain the tip amount and the total bill for a 15% tip, as in:

Selecting a Custom Tip Percentage

Next, use the `Slider` to specify a custom tip percentage. Drag the `Slider`'s thumb until the percentage reads 20%. As you drag the thumb, the tip percentage in the `Label` to the `Slider`'s left updates continuously. By default, a `Slider` allows you to select floating-point values from 0.0 to 100.0, but this application restricts the `Slider` to whole numbers from 0 to 30. Click the Calculate `Button` to display the updated tip and total:

15.5.2 Technologies Overview

This section introduces the technologies you'll use to build the Tip Calculator application.

Class `Application`

The class that launches a JavaFX application (which we'll present in

Section 15.5.4) is a subclass of `Application` (package `javafx.application`). When you execute the subclass:

- Method `main` calls **class `Application`'s `static` `launch` method** to begin executing the application.
- The `launch` method, in turn, causes the JavaFX runtime to create an object of the `Application` subclass and call its **`start` method**.
- The `Application` subclass's `start` method creates the GUI, attaches it to a `Scene` and places it on the `Stage` that `start` receives as an argument.

Arranging JavaFX Components with a `GridPane`

Recall that layout containers arrange JavaFX components in a `Scene`. A **`GridPane`** (package `javafx.scene.layout`) arranges JavaFX components in a rectangular grid of rows and columns. This application uses a `GridPane` to arrange views into five rows and two columns:

Each cell in a `GridPane` can be empty or hold one or more JavaFX components, including layout containers that arrange other controls. Each component in a `GridPane` can span multiple rows or columns, though we did not use that capability in this GUI. When you drag a `GridPane` onto Scene Builder's content panel, Scene Builder creates the `GridPane` with two columns and three rows by default. You can add and remove columns and rows as necessary. We'll discuss other `GridPane` features as we present the GUI-building steps. To learn more about class `GridPane`, visit:

Click here to view code image

```
https://openjfx.io/javadoc/23/javafx.graphics/javafx/scene/layout/
   GridPane.html
```

Creating and Customizing the GUI with Scene Builder

In Section 15.5.3, you'll create `Label`s, `TextField`s, a `Slider` and a `Button`:

- A `TextField` (package `javafx.scene.control`) can accept text input from the user or display text. You'll use one editable `TextField` to input the bill amount from the user and two uneditable `TextField`s to display the tip and total

amounts.

- A **`Slider`** (package `javafx.scene.control`) represents a value in the range 0.0–100.0 by default and allows the user to select a number in that range by moving the `Slider`'s thumb. You'll customize the `Slider` so the user can choose a custom tip percentage only from the more limited integer range 0–30.
- A **`Button`** (package `javafx.scene.control`) allows the user to initiate an action—in this application, clicking the Calculate `Button` calculates and displays the tip and total amounts.

Formatting Numbers as Locale-Specific Currency and Percentage Strings

You'll use **class `NumberFormat`** (package **`java.text`**) to create locale-specific currency and percentage strings—an important part of internationalization.

Event Handling

A user interacts with an application's GUI to indicate the tasks the application should perform. For example, when you write an e-mail, clicking the e-mail application's Send button tells the application to send the e-mail to the specified recipient(s).

GUIs are **event-driven**. When the user interacts with a GUI component, the interaction—known as an event—drives the program to perform an action. Some common user interactions that cause an application to perform a task include

- clicking a button,
- entering text in a text field,
- selecting an item from a menu,
- minimizing, maximizing or closing a window,
- pressing a mouse button and
- moving the mouse.

The code that performs a task in response to an event is called an event handler, and the process of responding to events is known as **event handling**. Before an application can respond to an event for a particular control, you must:

- Define an event handler that implements an appropriate interface—known as an **event-listener interface**.
- Indicate that an object of that class should be notified when the event occurs— known as **registering the event handler**.

In this application, you'll respond to two events—when the user moves the `Slider`'s thumb, the application will update the `Label` that displays the current tip percentage, and when the user clicks the Calculate `Button`, the application will calculate and display the tip and total bill amount.

We'll demonstrate that for certain events—such as when the user clicks a `Button`—you can link a control to its event-handling method (that is, register the event handler) by using the Code section of Scene Builder's Inspector window. In this case, JavaFX implements the event-listener interface for you and it calls the method you specify. For events that occur when the value of a control's property changes, such as when the user moves a `Slider`'s thumb to change the `Slider`'s value, you must create the event handler entirely in code, as we'll demonstrate.

Implementing Interface `ChangeListener` for Handling `Slider` Thumb Position Changes

You'll implement **interface `ChangeListener`** (package `javafx.beans.value`) to respond when the user moves the `Slider`'s thumb. In particular, you'll use the interface's `changed` method to display the updated tip percentage as the user moves the `Slider`'s thumb. `ChangeListener` is a functional interface, so we'll use a lambda to implement it.

Model-View-Controller (MVC) Architecture

JavaFX programs in which the GUI is implemented as FXML adhere to the **Model-View-Controller (MVC) design pattern**, which separates an application's data (contained in the **model**) from the application's GUI (the **view**) and the application's processing logic (the **controller**).

The controller implements logic for processing user inputs. The model contains the program's data, and the view presents the data stored in the model. When a user provides some input, the controller modifies the model with the given input. In the Tip Calculator, the model is the bill amount, the tip and the total. When the model changes, the controller updates the view to present the

changed data.

In a JavaFX FXML application, a controller class defines instance variables for interacting with controls programmatically, as well as event-handling methods that respond to the user's interactions. The controller class may also declare additional instance variables, `static` variables and methods that support the application's operation. In a simple application like the Tip Calculator, the model and controller are often combined into a single class, as we'll do in this example.

`FXMLLoader` Class

When a JavaFX FXML application begins executing, **class `FXMLLoader`'s `static` method `load`** reads the contents of the GUI's FXML file into memory. Then, `load` performs the following actions:

- Creates the scene graph containing the GUI's layouts and controls and returns a **`Parent` (package `javafx.scene`) reference to the scene graph's root node**.
- Creates an instance of the controller class and initializes the controller's instance variables for the components that will be manipulated programmatically.
- Creates and registers the event handlers for any events specified in the FXML.

We'll discuss these steps in more detail in Sections 15.5.4 and 15.5.5.

15.5.3 Building the Application's GUI

In this section, we'll present the precise steps for creating the Tip Calculator's GUI.

fx:id Property Values for This Application's Controls

If the controller class will manipulate a control or layout programmatically (as we'll do with one `Label`, all the `TextField`s and the `Slider`), you must provide a name for that control or layout. In Section 15.5.4, you'll declare Java variables for each such component in the FXML, and we'll discuss how those variables are initialized for you. Each object's name is specified via its **fx:id property**. You can set this property's value by selecting a component in your scene, then expanding the Inspector window's

Code section—the fx:id property appears at the top of that section. The following diagram contains the fx:id properties of the Tip Calculator's programmatically manipulated controls. Our naming convention is to use the control's class name in the fx:id property. So, the fx:ids `amountTextField`, `tipTextField` and `totalTextField` each include the control's class name—`TextField`:

Creating the `TipCalculator.fxml` File

As you did in Section 15.4.1, open Scene Builder to create a new empty FXML file. Then, select File > Save to display the Save As dialog:

- specify the location in which to store the file (the same folder where you'll store the application's Java source code),
- name the file `TipCalculator.fxml` and
- click the Save button.

Step 1: Adding a `GridPane`

Drag a `GridPane` from the Library window's Containers section onto Scene Builder's content panel. By default, the `GridPane` contains two columns and three rows:

Step 2: Adding Rows to the `GridPane`

Recall that the application's final GUI has two columns and five rows. Here, you'll add two more rows. To add a row above or below an existing row:

- Right-click any row's row number tab and select either Grid Pane > Add Row Above or Grid Pane > Add Row Below.
- Repeat this process to add another row.

After adding two rows, the `GridPane` should appear as follows:
You can use similar steps to add columns. You can delete a row or column by right-clicking the tab containing its row or column number and selecting Delete.

Step 3: Adding the Controls to the `GridPane`

You'll now add the controls to the `GridPane`. When you drag a control onto the `GridPane`:

- If the control has an fx:id in the following diagram, use the corresponding variable name to set the control's fx:id property in the Inspector window's Code section.
- If the control displays text in the following diagram, use the corresponding text to set the control's Text property in the Inspector window's Properties section.

Perform the following steps to build the GUI:

- ***Adding the `Label`s.*** Drag `Label`s from the Library window's Controls section into the first four rows of column 0 (the `GridPane`'s left column). As you add each, set its text.
- ***Adding the `TextField`s.*** Drag `TextField`s from the Library window's Controls section into rows 0, 2 and 3 of column 1 (the `GridPane`'s right column).
- ***Adding a `Slider`.*** Drag a horizontal `Slider` from the Library window's Controls section into row 1 of column 1.
- ***Adding a `Button`.*** Drag a `Button` from the Library window's Controls section into row 4 of column 1. Change the `Button`'s text to Calculate. You can set the `Button`'s text by double-clicking it or by selecting the `Button` then setting its Text property in the Inspector window's Properties section.

The `GridPane` should appear as follows:

Step 4: Sizing the `GridPane` to Fit Its Contents

When you begin designing a GUI by adding a layout, Scene Builder automatically sets the layout object's Pref Width property to `600` and Pref Height property to `400`, which is much larger than this GUI's final width and height. For this application, we'd like the layout's size to be computed based on the layout's contents. To make this change:

- First, select the `GridPane` by clicking inside the `GridPane`, but not on any of the controls you've placed into its columns and rows. Sometimes, it's easier to select the `GridPane` node in the Scene Builder Document window's Hierarchy section in Scene Builder's left column.
- In the Inspector's Layout section, reset the Pref Width and Pref Height property values to their defaults (as you did in Section 15.4.6). This sets both properties' values to `USE_COMPUTED_SIZE`, so the layout calculates its own size.

The layout now appears as follows:

Step 5: Right-Aligning `GridPane` Column 0's Contents

A `GridPane` column's contents are left-aligned by default. To right-align the contents of column 0, select it by clicking the tab at the top or bottom of the column, then in the Inspector's Layout section, set the Halignment (horizontal alignment) property to `RIGHT`.

Step 6: Sizing the `GridPane` Columns to Fit Their Contents

By default, Scene Builder sets each `GridPane` column's width to 100 pixels and each row's height to 30 pixels to ensure that you can easily drag controls into the `GridPane`'s cells. In this application, we sized each column to fit its contents. To do so, select the column 0 by clicking the tab at the top or bottom of the column, then in the Inspector's Layout section, reset the Pref Width property to its default size (that is, `USE_COMPUTED_SIZE`) to indicate that the column's width should be based on its widest child—the Amount `Label` in this case. Repeat this process for column 1. The `GridPane` should appear as follows:

Step 7: Sizing the `Button`

By default, Scene Builder sets a `Button`'s width based on its text. For this application, we chose to make the `Button` the same width as the other controls in the `GridPane`'s right column. To do so, select the `Button`, then in the Inspector's Layout section, set the Max Width property to `MAX_VALUE`. This causes the `Button`'s width to grow to fill the column's width.

Previewing the GUI

Preview the GUI by selecting Preview > Show Preview in Window: Notice that there's no space between the `Label`s in the left column and the controls in the right column, nor is there space around the `GridPane`. By default, the `Stage` is sized to fit the `Scene`'s contents, so many of the controls touch the window's borders. You'll fix these issues in the next step.

Step 8: Configuring the `GridPane`'s Padding and Horizontal Gap Between Its Columns

The space between a node's contents and its top, right, bottom and left edges is known as **padding**, which separates the contents from the node's edges. Since the `GridPane`'s size determines the `Stage`'s window size, the `GridPane`'s padding separates its children from the window's edges. To set the `GridPane`'s padding, select it, then in the Inspector's Layout section, set the Padding property's four values (which represent the TOP, RIGHT, BOTTOM and LEFT) to `14`—a commonly used distance between a control's edge and the `Scene`'s edge.

You can specify the default amount of space between a `GridPane`'s columns and rows with its Hgap (horizontal gap) and Vgap (vertical gap) properties, respectively. Because Scene Builder sets each `GridPane` row's height to 30 pixels—which is greater than the heights of this application's controls—there's already some vertical space between the components. To specify the horizontal gap between the columns, select the `GridPane` in the Document window's Hierarchy section, then in the Inspector's Layout section, set the Hgap property to `8`—a commonly used distance between controls. If you'd like to precisely control the vertical space between components, you can reset each row's Pref Height to its default value, then set the `GridPane`'s Vgap property.

Step 9: Making the `tipTextField` and `totalTextField` Uneditable and Not Focusable

The `tipTextField` and `totalTextField` are used in this application only to display results, not receive text input. For this reason, they should not be interactive. You can enter text in a `TextField` only if it's "in **focus**"—that is, it's the control the user is interacting with. When you click an interactive control, it receives the focus. Similarly, when you press the *Tab* key, the focus transfers from the current focusable control to the next one— by default, this is the order in which the controls were added to the GUI. Interactive controls—such as `TextField`s, `Slider`s and `Button`s—are focusable by default. Non-interactive controls—like `Label`s—are not focusable.

In this application, the `tipTextField` and `totalTextField` are neither editable nor focusable. To make these changes, select both `TextField`s, then in the Inspector's Properties section uncheck the Editable and Focus Traversable properties. To select multiple controls at once, you can click the first (in the Document window's Hierarchy section or in the content panel), then hold the *Shift* key and click each of the others.

Step 10: Setting the `Slider`'s Properties

To complete the GUI, you'll now configure the Tip Calculator's `Slider`. By default, a `Slider`'s range is `0.0` to `100.0`, and its initial value is 0.0. This application allows only integer tip percentages in the range 0 to 30 with a default of 15. To make these changes, select the `Slider`, then in the Inspector's Properties section, set the `Slider`'s Max property to `30` and the Value property to `15`. We also set the Block Increment property to `5`—this is the amount by which the Value property increases or decreases when the user clicks between an end of the `Slider` and the `Slider`'s thumb. Save the FXML file by selecting File > Save.

Though we set the Max, Value and Block Increment properties to integer values, the `Slider` still produces floating-point values as the user moves its thumb. In the application's Java code, we'll restrict the `Slider`'s values to integers when we respond to its events.

Previewing the Final Layout

You've now completed the Tip Calculator's design. Select Preview > Show Preview in Window to view the final GUI:

When we discuss the `TipCalculatorController` class in Section 15.5.5, we'll show how to specify the Calculate `Button`'s event handler in the FXML file.

Specifying the Controller Class's Name

As we mentioned in Section 15.5.2, in a JavaFX FXML application, the application's controller class typically defines instance variables for interacting with controls programmatically, as well as event-handling methods. To ensure that an object of the controller class is created when the application loads the FXML file at runtime, you must specify the controller class's name in the FXML file:

- Expand the Scene Builder Document window's Controller section—this is located below the Hierarchy section in the bottom-left corner of Scene Builder.
- In the Controller Class field, type `TipCalculatorController`—by convention, the controller class's name starts with the same name as the FXML file (in this case, `TipCalculator`) and ends with `Controller`.

Specifying the Calculate `Button`'s Event-Handler

Method Name

You can specify in the FXML file the names of the methods that handle the controls' events. When you select a control, the Inspector window's Code section shows all the events for which you can specify event handlers. When the user clicks a `Button`, the method in the On Action field is called. This method is defined in the controller class you specify in Scene Builder's Controller window. Enter `calculateButtonPressed` in the `Button`'s On Action field.

Generating a Sample Controller Class

You can have Scene Builder generate the initial controller class containing the variables you'll use to interact with controls programmatically and the empty Calculate `Button` event handler. Scene Builder calls this the "controller skeleton." Select View > Show Sample Controller Skeleton to generate the skeleton: Notice that the sample class has the class name you specified, a variable for each control that has an fx:id and an empty Calculate `Button` event handler. We'll discuss the `@FXML` annotation in Section 15.5.5. To use this skeleton to create your controller class, click the Copy button, then paste the contents into a file named `TipCalculatorController.java` in the same folder as the `TipCalculator.fxml` file you created in this section.

15.5.4 `TipCalculator` Class

A simple JavaFX FXML-based application has two Java source-code files. For the Tip Calculator application, these are:

- `TipCalculator.java`—This file contains the `TipCalculator` class (discussed in this section), which declares the `main` method that loads the FXML file to create the GUI and attaches the GUI to a `Scene` displayed on the application's `Stage`.
- `TipCalculatorController.java`—This file contains the controller class (discussed in Section 15.5.5), where you'll specify the `Slider` and `Button` controls' event handlers.

Figure 15.1 presents class `TipCalculator`. As we discussed in Section 15.5.2, the starting point for a JavaFX application is an `Application` subclass, so class `TipCalculator` extends `Application` (line 9). The `main` method calls class `Application`'s `static launch`

method (line 23) to initialize the JavaFX runtime and to begin executing the application. This method causes the JavaFX runtime to create an object of the `TipCalculator` class and calls its `start` method (lines 10–19), passing the `Stage` object representing the window in which the application will be displayed. The JavaFX runtime creates the window.

Click here to view code image

```
1  // Fig. 15.1: TipCalculator.java
2  // Main application class that loads and displays the Tip Calculator's GUI
3  import javafx.application.Application;
4  import javafx.fxml.FXMLLoader;
5  import javafx.scene.Parent;
6  import javafx.scene.Scene;
7  import javafx.stage.Stage;
8
9  public class TipCalculator extends Application {
10    @Override
11    public void start(Stage stage) throws Exception {
12       Parent root =
13          FXMLLoader.load(getClass().getResource("TipCalculator.fxml"));
14
15       var scene = new Scene(root); // attach scene graph to scene
16       stage.setTitle("Tip Calculator"); // displayed in window's title bar
17       stage.setScene(scene); // attach scene to stage
18       stage.show(); // display the stage
19    }
20
21    public static void main(String[] args) {
22       // create a TipCalculator object and call its start method
23       launch(args);
24    }
25 }
```

Fig. 15.1 | Main application class that loads and displays the Tip Calculator's GUI.

Overridden `Application` Method `start`

Method `start` (lines 10–19) creates the GUI, attaches it to a `Scene` and places it on the `Stage` that method `start` receives as an argument. Lines 12–13 use class `FXMLLoader`'s `static` method `load` to create the GUI's scene graph. This method:

- Returns a `Parent` (package `javafx.scene`) reference to the scene graph's root node—this is a reference to the GUI's

`GridPane` in this application.

- Creates an object of the `TipCalculatorController` class that we specified in the FXML file.
- Initializes the controller's instance variables for the components that are manipulated programmatically.
- Registers the event handlers specified in the FXML with the appropriate controls, enabling them to call their event handlers in response to user interactions.

We discuss the initialization of the controller's instance variables and the registration of the event handlers in Section 15.5.5.

Creating the `Scene`

To display the GUI, you must attach it to a `Scene`, then attach the `Scene` to the `Stage` that method `start` receives as an argument. To attach the GUI to a `Scene`, line 15 creates a `Scene`, passing `root` (the scene graph's root node) as an argument to the constructor. By default, the `Scene`'s size is determined by the size of the scene graph's root node. Overloaded versions of the `Scene` constructor allow you to specify the `Scene`'s size and fill (a color, gradient or image), which appears in the `Scene`'s background. Line 16 uses `Stage` method `setTitle` to specify the text that appears in the `Stage` window's title bar. Line 17 calls `Stage` method `setScene` to place the `Scene` onto the `Stage`. Finally, line 18 calls `Stage` method `show` to display the `Stage` window.

15.5.5 `TipCalculatorController` Class; Implementing an Event Handler with a Lambda

Figure 15.2 presents the `TipCalculatorController` class that responds to user interactions with the application's `Button` and `Slider`. We split the class into pieces for discussion purposes. Lines 3–10 show the class's `import` statements:

Click here to view code image

```
1  // Fig. 15.2: TipCalculatorController.java
2  // Controller that handles calculateButton and tipPercentageSlider events
3  import java.math.BigDecimal;
4  import java.math.RoundingMode;
```

```
5  import java.text.NumberFormat;
6  import javafx.event.ActionEvent;
7  import javafx.fxml.FXML;
8  import javafx.scene.control.Label;
9  import javafx.scene.control.Slider;
10 import javafx.scene.control.TextField;
11
```

Fig. 15.2 | `TipCalculatorController`'s `import` declarations.

The classes and interfaces used by class `TipCalculatorController` include:

- Class `BigDecimal` (line 3; introduced in Section 4.11) is used to perform precise monetary calculations. The `RoundingMode enum` of package `java.math` (line 4) is used to specify how `BigDecimal` values are rounded during calculations or when formatting floating-point numbers as `String`s.
- Class `NumberFormat` of package `java.text` (line 5) provides numeric formatting capabilities, such as locale-specific currency and percentage formats. For example, in the U.S. locale, the monetary value 34.95 is formatted as $34.95 and the percentage 15 is formatted as 15%. Class `NumberFormat` determines the locale of the system on which your application runs then formats currency amounts and percentages accordingly.
- A `Button`'s event handler receives an `ActionEvent` object (line 6; package `javafx.event`) indicating which `Button` the user clicked. Many JavaFX controls support `ActionEvent`s.
- The annotation `FXML` (line 7; package `javafx.fxml`) is used in a JavaFX controller class's code (as `@FXML`) to mark instance variables that should refer to JavaFX components in the GUI's FXML file and methods that can respond to the events of JavaFX components in the GUI's FXML file.
- Package `javafx.scene.control` (lines 8–10) contains many JavaFX control classes, including `Label`, `Slider` and `TextField`.

`TipCalculatorController`'s `static` Variables and Instance Variables

Lines 14–35 declare class `TipCalculatorController`'s `static` and instance variables. The `NumberFormat` objects (lines 14–17) are used to format currency values and percentages, respectively.

`NumberFormat` method `getCurrencyInstance` returns a `NumberFormat` object that formats values as currency using the default locale for the system on which the application is running. Similarly, **`NumberFormat` method `getPercentInstance`** returns a `NumberFormat` object that formats values as percentages using the system's default locale. The `BigDecimal` object `tipPercentage` (line 19) stores the current tip percentage and is used in the tip calculation when the user clicks the Calculate `Button`.

Click here to view code image

```
12  public class TipCalculatorController {
13     // formatters for currency and percentages
14     private static final NumberFormat currency =
15        NumberFormat.getCurrencyInstance();
16     private static final NumberFormat percent =
17        NumberFormat.getPercentInstance();
18
19     private BigDecimal tipPercentage = new BigDecimal("0.15"); // 15%
20
21     // GUI controls defined in FXML and used by the controller's code
22     @FXML
23     private TextField amountTextField;
24
25     @FXML
26     private Label tipPercentageLabel;
27
28     @FXML
29     private Slider tipPercentageSlider;
30
31     @FXML
32     private TextField tipTextField;
33
34     @FXML
35     private TextField totalTextField;
36
```

`@FXML` Annotation

Recall from Section 15.5.3 that each control that this application manipulates in its Java source code needs an fx:id. Lines 22–35 declare the controller class's corresponding instance variables. The **`@FXML` annotation** that precedes each declaration (lines 22, 25, 28, 31 and 34) indicates that the variable name can be used in the FXML file that describes the application's GUI. The variable names that you specify in the controller class must precisely match the fx:id values you specified when building the GUI; otherwise, the

`FXMLLoader` will throw a `LoadException`. When the `FXMLLoader` loads `TipCalculator.fxml` to create the GUI, it also initializes each of the controller's instance variables declared with `@FXML` to ensure that they refer to the corresponding GUI components.

`TipCalculatorController`'s `calculateButtonPressed` Event Handler

Lines 38–53 declare `TipCalculatorController`'s `calculateButtonPressed` method, which is called when the user clicks the Calculate `Button`. The `@FXML` annotation (line 38) preceding the method indicates that this method can be used to specify a control's event handler in the FXML file that describes the application's GUI. For a control that generates an `ActionEvent` (as many JavaFX controls do), the event-handling method must return `void` and receive one `ActionEvent` parameter (line 39).

Click here to view code image

```
37      // calculates and displays the tip and total amounts
38      @FXML
39      private void calculateButtonPressed(ActionEvent event) {
40         try {
41            BigDecimal amount = new BigDecimal(amountTextField.getText());
42            BigDecimal tip = amount.multiply(tipPercentage);
43            BigDecimal total = amount.add(tip);
44
45            tipTextField.setText(currency.format(tip));
46            totalTextField.setText(currency.format(total));
47         }
48         catch (NumberFormatException ex) {
49            amountTextField.setText("Enter amount");
50            amountTextField.selectAll();
51            amountTextField.requestFocus();
52         }
53      }
54
```

Registering the Calculate Button's Event Handler

When the `FXMLLoader` loads `TipCalculator.fxml` to create the GUI, it creates and registers an event handler for the Calculate `Button`'s `ActionEvent`. The event handler for this event must implement interface **`EventHandler<ActionEvent>`** (**package `javafx.event`**)—`EventHandler` is a generic interface containing a **`handle` method** that returns `void` and receives an `ActionEvent` parameter. This

method, in turn, calls `calculateButtonPressed` when the user clicks the Calculate `Button`. `FXMLLoader` performs similar tasks for every event listener you specify via the Scene Builder Inspector window's Code section.

Calculating and Displaying the Tip and Total Amounts

Lines 41–46 calculate and display the tip and total. Line 41 calls the `amountTextField`'s **`getText` method** to get the bill amount entered by the user. This `String` is passed to `BigDecimal`'s constructor, which throws a `NumberFormatException` if its argument is not a number. In that case, line 49 calls `amountTextField`'s **`setText` method** to display `"Enter amount"` in the `TextField`. Line 50 calls `amountTextField`'s **`selectAll` method** to select the `TextField`'s text. Line 51 calls `amountTextField`'s **`requestFocus` method** to give the `TextField` the focus. Now, the user can enter a new value in the `amountTextField` without having to first select the text. `TextField` inherits methods `getText`, `setText` and `selectAll` from class `TextInputControl` (package `javafx.scene.control`) and method `requestFocus` from class `Node` (package `javafx.scene`).

If line 41 does not throw an exception, line 42 calculates the `tip` by calling method `multiply` to multiply the `amount` by the `tipPercentage`, and line 43 calculates the `total` by adding the `tip` to the bill `amount`. Next, lines 45 and 46 use the `currency` object's **`format` method** to create currency-formatted `String`s representing the tip and total amounts, which we display in `tipTextField` and `totalTextField`, respectively.

`TipCalculatorController`'s `initialize` Method

Lines 56–69 declare `TipCalculatorController`'s `private initialize` method. When present and annotated with `@FXML`, JavaFX will call this method to configure the controller before the GUI is displayed. Line 59 calls the `currency` object's `setRoundingMode` method to specify that currency values should be rounded using the `RoundingMode.HALF_EVEN` (as discussed in Section 4.11).

Click here to view code image

```
55      // called by FXMLLoader to initialize the controller
56      @FXML
57      private void initialize() {
```

```
58          // 0-4 rounds down, 5-9 rounds up
59          currency.setRoundingMode(RoundingMode.HALF_EVEN);
60
61          // listener for changes to tipPercentageSlider's value
62          tipPercentageSlider.valueProperty().addListener(
63             (ov, oldValue, newValue) -> {
64                tipPercentage =
65                   BigDecimal.valueOf(newValue.intValue() / 100.0);
66                tipPercentageLabel.setText(percent.format(tipPercentage));
67             }
68          );
69       }
70    }
```

Using a Lambda for Event Handling

Each JavaFX control has properties. Some—such as a `Slider`'s value—can generate events when they change. For such events, you must manually register as the event handler an object that implements the `ChangeListener` interface (package `javafx.beans.value`).

`ChangeListener` is a generic type that's specialized with the property's type. The call to `valueProperty` (line 62) returns a `DoubleProperty` (package `javax.beans.property`) that represents the `Slider`'s value. A `DoubleProperty` is an **`ObservableValue<Number>`** that can notify listeners when a value changes. Each class that implements interface `ObservableValue` provides **method `addListener`** (called in lines 62–68) to register an event-handler that implements interface `ChangeListener`. For a `Slider`'s value, `addListener`'s argument is an object that implements `ChangeListener<Number>` because the `Slider`'s value is a numeric value.

`ChangeListener` is a generic functional interface, so we can conveniently implement it using a lambda (lines 63–67; introduced in Section 14.3). **`ChangeListener`'s `abstract` `changed` method** receives three arguments:

- an `ObservableValue<T>`,
- the old value of type `T`, and
- the new value of type `T`.

Since the `Slider`'s `addListener` method expects a `ChangeListener<Number>` object, the compiler infers that `T` is `Number` for the lambda's three parameters:

- `ov`—the `ObservableValue<Number>` that changed,
- `oldValue`—a `Number` containing the `Slider`'s old value before the event occurred and
- `newValue`—a `Number` containing the `Slider`'s new value.

When the user moves the `Slider`'s thumb, lines 64–65 store the new tip percentage and line 66 updates the `tipPercentageLabel`.

Generative AI

- Prompt genAIs asking for a list of JavaFX spacing and layout conventions.
- Prompt genAIs asking how JavaFX's designers arrived at the recommended spacing conventions.

15.6 Features Covered in the Other JavaFX Chapters

JavaFX is a robust GUI, graphics and multimedia technology. In Chapters 16 and 17, you'll:

- Learn additional JavaFX layouts and controls.
- Handle other event types (such as `MouseEvent`s).
- Apply transformations (such as moving, rotating, scaling and skewing) and effects (such as drop shadows, blurs, reflection and lighting) to a scene graph's nodes.
- Use CSS to specify the look-and-feel of controls.
- Use JavaFX properties and data binding to enable automatic updating of controls as corresponding data changes.
- Access files and folders on disk using `FileChooser` and `DirectoryChooser` dialogs.
- Use JavaFX graphics capabilities.
- Perform JavaFX animations.
- Use JavaFX multimedia capabilities to play audio and video.

15.7 Wrap-Up

In this chapter, we introduced JavaFX for creating graphical user interfaces (GUIs). We briefly discussed JavaFX's benefits over the older AWT and Swing GUI technologies. We explored a JavaFX application window's structure, including its stage, scene and various controls. We explained key JavaFX concepts such as using Scene Builder to create the FXML that declaratively specifies a GUI's layout and controls, the Model-View-Controller (MVC) design pattern, and how a controller class responds to the events generated by users interacting with GUI controls.

We used two examples to introduce key JavaFX concepts. In the simple Welcome application, we used drag-and-drop visual programming techniques in Scene Builder to design a GUI consisting of a `VBox` layout container, a `Label` and an `ImageView` without writing any Java code.

In the Tip Calculator application, we used Scene Builder to create a more elaborate GUI consisting of `Label`s, `TextField`s, a `Slider` and a `Button` arranged in a `GridPane` layout container with multiple rows and columns. We used class `FXMLLoader` to read the FXML created in Scene Builder, create the corresponding GUI, create an instance of the controller class and attach the GUI controls to the controller class's `@FXML` annotated fields and methods.

The Tip Calculator's controller class responded to user interactions with the application's `Button` and `Slider` controls and updated the GUI accordingly. We showed that some event handlers (such as the `Button`'s On Action handler) can be specified directly in Scene Builder as part of the FXML, but event handlers for changes to a control's property values (such as the `Slider`'s value) must be implemented directly in the controller's code. We used a lambda to implement the event handler that responded to the `Slider`'s value changes. Throughout the examples, we emphasized JavaFX best practices, such as proper spacing and alignment of GUI elements.

In the next chapter, you'll learn more details of laying out nodes in a scene graph with various JavaFX layout panels so you can build more intricate GUIs. We'll demonstrate additional JavaFX controls and event types. We'll use CSS to style GUIs. We'll customize a control's display. You'll also learn more about JavaFX properties and how to use data binding to automatically update elements in a GUI with new data.

Special Section: Generative AI Exercises

For each of the following questions, try multiple genAIs to

determine which performs best. Always verify the results. Keep in mind that they sometimes make mistakes and even hallucinate, creating unreliable information.

15.1 ***(Generative AI: JavaFX Demos)*** Prompt genAIs to locate JavaFX demo websites with applications that show JavaFX's range of capabilities.

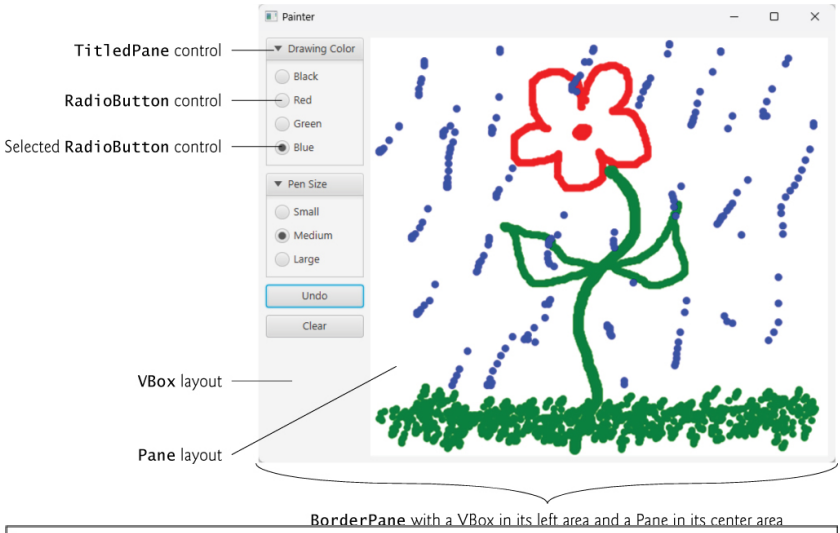
15.2 ***(Generative AI: JavaFX FXML Scrapbook Layout)*** Prompt genAIs with the names of four image files, asking them to design a JavaFX FXML layout that displays each image in a 3-by-2 ratio rotated at different angles, like you might find in a scrapbook. Ask them to provide a link to download the FXML file. Place the file in a folder with your images, then open the FXML file with Scene Builder and preview it as you did with the Welcome app in Section 15.4.

15.3 ***(Generative AI: JavaFX Tip Calculator User Interface Refactoring)*** Prompt genAIs with the Tip Calculator application's FXML file, asking them to refactor the JavaFX user interface to change the `tipPercentageSlider` from horizontal to vertical. Ask them to place the `tipPercentageLabel` and the `tipPercentageSlider` in a new column at the GUI's right side, with the `tipPercentageLabel` at the top and the `tipPercentageSlider` occupying the remainder of the right column. Open the refactored user interface in Scene Builder, then run the Tip Calculator application with the new user interface.

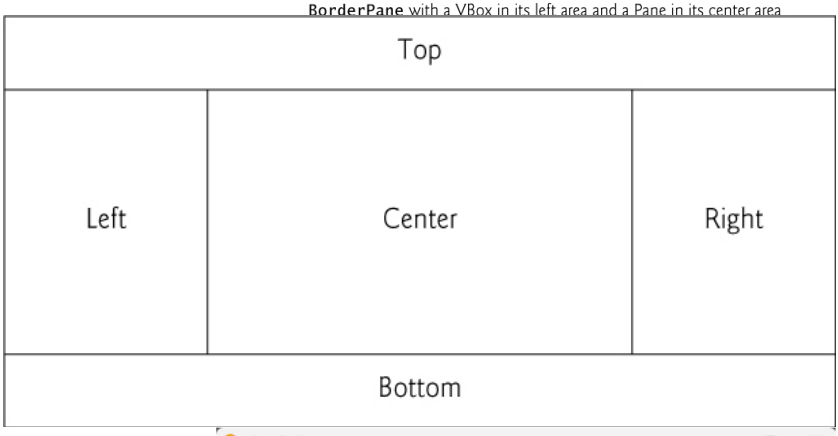
15.4 ***(Generative AI: Identifying a JavaFX Application's Model, View and Controller)*** Prompt genAIs with Section 15.5's Tip Calculator application FXML and Java source-code files. Ask the genAIs to identify the components of the Model-View-Controller (MVC) design pattern in the application.

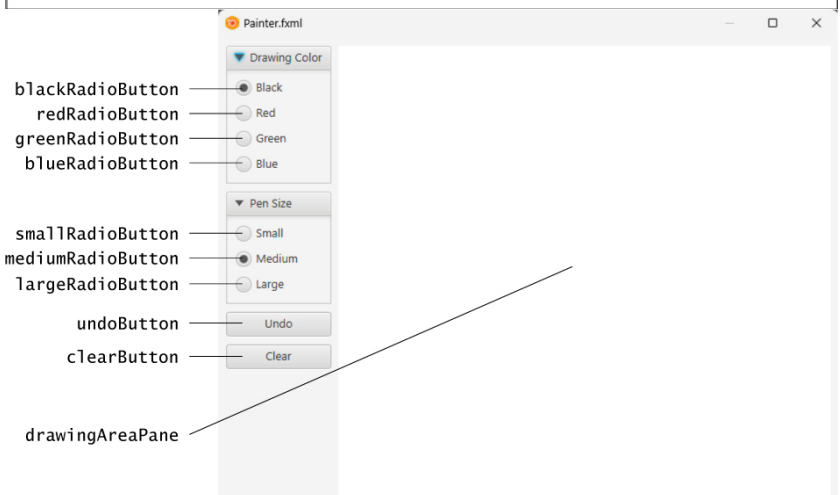
15.5 ***(Generative AI: JavaFX `Node` Transformations)*** Prompt genAIs, asking for a complete list of JavaFX `Node` transformations with brief explanations.

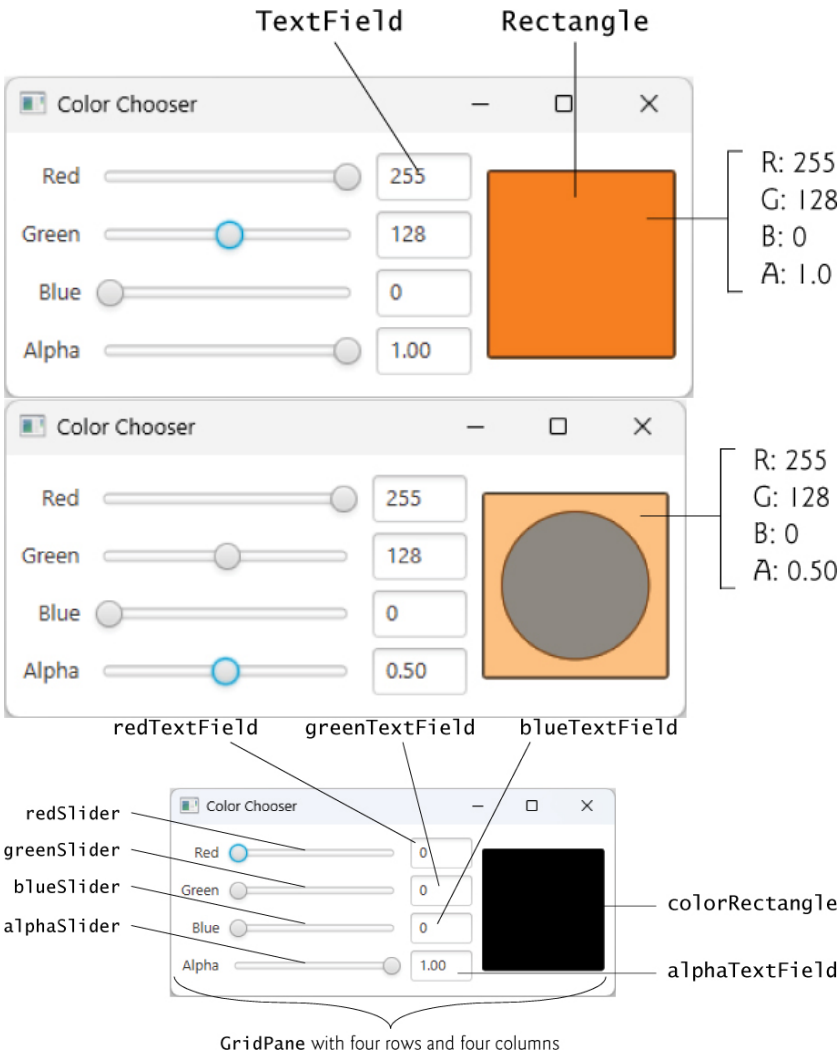
15.6 ***(Generative AI: JavaFX Effects)*** Prompt genAIs, asking for a complete list of JavaFX effects with brief explanations.

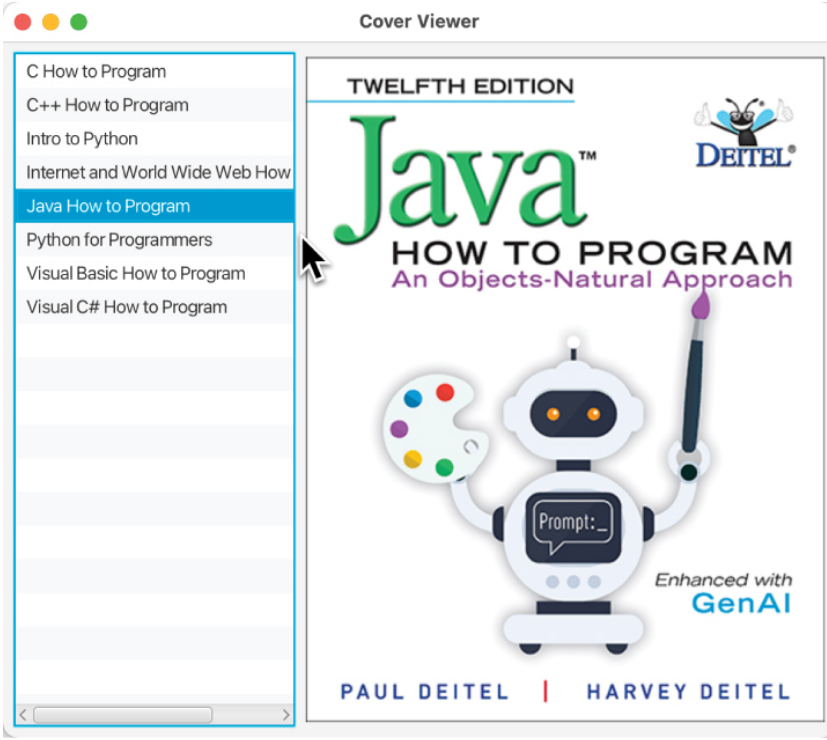


BorderPane with a VBox in its left area and a Pane in its center area





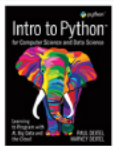

TextField
Rectangle
Color Chooser
Red
Green
Blue
Alpha
255
128
0
1.00
R: 255
G: 128
B: 0
A: 1.0
Color Chooser
Red
Green
Blue
Alpha
255
128
0
0.50
R: 255
G: 128
B: 0
A: 0.50
redTextField
greenTextField
blueTextField
redSlider
greenSlider
blueSlider
alphaSlider
Color Chooser
Red
Green
Blue
Alpha
0
0
0
1.00
colorRectangle
alphaTextField
GridPane with four rows and four columns


Cover Viewer
C How to Program
C++ How to Program
Intro to Python
Internet and World Wide Web How
Java How to Program
Python for Programmers
Visual Basic How to Program
Visual C# How to Program
TWELFTH EDITION
Java™
DEITEL®
HOW TO PROGRAM
An Objects-Natural Approach
Prompt:_
Enhanced with
GenAI
PAUL DEITEL | HARVEY DEITEL

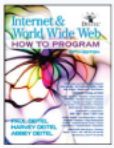
booksListView
coverImageView
Cover Viewer
C How to Program
C++ How to Program
Intro to Python
Internet and World Wide Web How
Java How to Program
Python for Programmers
Visual Basic How to Program
Visual C# How to Program
TWELFTH EDITION
Java™
HOW TO PROGRAM
An Objects-Natural Approach
DEITEL®
Prompt:_
Enhanced with
GenAI
PAUL DEITEL | HARVEY DEITEL
BorderPane

Cover Viewer

C++ How to Program

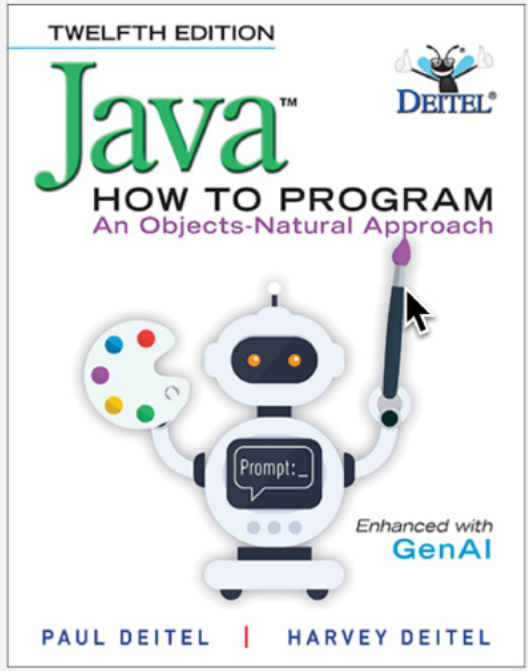


Intro to Python



Internet and World Wide Web How to Program

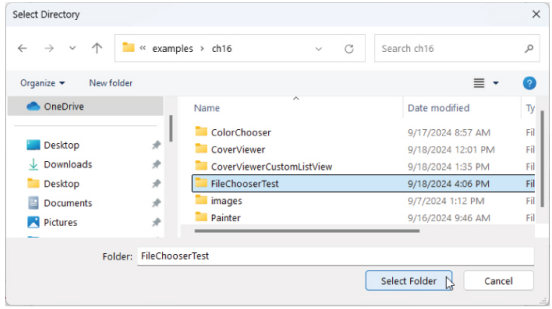
Java How to Program

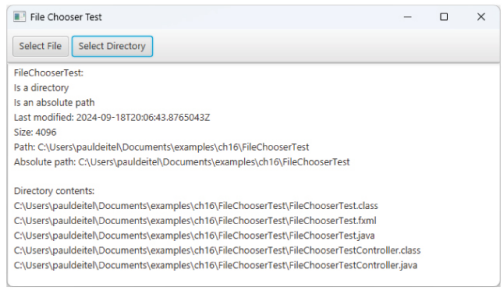


File Chooser Test

Select File | Select Directory

Select file or directory





JavaFX GUI: Part 2

Objectives

In this chapter, you'll:

- Learn more about laying out nodes in a scene graph with JavaFX layout panels.
- Continue building JavaFX GUIs with Scene Builder.

- Create and manipulate `RadioButton`s and `ListView`s.
- Use `BorderPane`s and `TitledPane`s to arrange controls.
- Handle mouse events.
- Use property binding and property listeners to perform tasks when a control's property value changes.
- Programmatically create layouts and controls.
- Customize a `ListView`'s cells with a custom cell factory.
- Use `FileChooser` and `DirectoryChooser` dialogs to enable the user to select a file or directory.
- See an overview of other JavaFX capabilities, updates and online resources.

Outline

16.1 Introduction

16.2 Laying Out Nodes in a Scene Graph

16.3 Painter Application: `RadioButton`s, Mouse Events and Shapes

16.3.1 Technologies Overview

16.3.2 Creating the `Painter.fxml` File

16.3.3 Building the GUI

16.3.4 `Painter` Subclass of `Application`

16.3.5 `PainterController` Class

16.4 Color Chooser Application: Property Bindings and Property Listeners

16.4.1 Technologies Overview

16.4.2 Building the GUI

16.4.3 `ColorChooser` Subclass of `Application`

16.4.4 `ColorChooserController` Class

16.5 Cover Viewer Application: Data-Driven GUIs with JavaFX Collections

16.5.1 Technologies Overview

16.5.2 Building the GUI

16.5.3 `CoverViewer` Subclass of `Application`

16.5.4 `CoverViewerController` Class

16.6 Cover Viewer Application: Customizing `ListView` Cells
16.6.1 Technologies Overview
16.6.2 Copying the `CoverViewer` Application
16.6.3 `ImageTextCell` Custom `ListView` Cell Layout
16.6.4 `CoverViewerController` Class
16.7 `FileChooser` and `DirectoryChooser` Dialogs
16.8 Other JavaFX Capabilities and JavaFX Accessibility
16.9 JavaFX Updates
16.10 JavaFX Resources and Libraries
16.11 Wrap-Up
Special Section: Generative AI Exercises

16.1 Introduction

This chapter continues our JavaFX presentation. In this chapter, you'll:

- Use additional layouts (`TitledPane`, `BorderPane` and `Pane`) and controls (`RadioButton` and `ListView`).
- Handle mouse and `RadioButton` events.
- Set up event handlers that respond to property changes on controls (such as the value of a `Slider`).
- Display `Rectangle`s and `Circle`s as nodes in the scene graph.
- Bind a collection of objects to a `ListView` that displays the collection's contents.
- Customize the appearance of a `ListView`'s cells.
- Use `FileChooser` and `DirectoryChooser` dialogs to enable the user to select a file or directory.

Finally, we overview other JavaFX capabilities, updates and online resources.

16.2 Laying Out Nodes in a Scene Graph

A layout determines the size and positioning of nodes in the scene graph.

Node Size

In general, a node's size should not be defined explicitly. Doing so often creates a design that looks pleasing when it loads but deteriorates as the application is resized or the content updates. In addition to `width` and `height`, most JavaFX nodes have additional properties that specify a node's range of acceptable sizes as it's laid out within its parent node:

- The **`minWidth`** and **`minHeight`** **properties** specify a node's minimum size in pixels.
- The **`maxWidth`** and **`maxHeight`** **properties** specify a node's maximum size in pixels.
- The **`prefWidth`** and **`prefHeight`** **properties** specify a node's preferred size that should be used by the layout in most cases.

Node Position and Layout Panes

A node's position should be defined relative to its parent node and the other nodes in its parent. JavaFX **layout panes** are container nodes that arrange their child nodes in a scene graph relative to one another based on their sizes and positions. Child nodes are controls, other layout panes, shapes and more.

Most JavaFX layout panes use relative positioning—if a layout-pane node is resized, it adjusts its children's sizes and positions accordingly, based on their preferred, minimum and maximum sizes. The following table describes each of the JavaFX layouts, including those presented in Chapter 15—these are defined in the packages `javafx.scene.layout` and `javafx.scene.controls`. In this chapter, we'll use `Pane`, `BorderPane`, `GridPane` and `VBox` from the `javafx.scene.layout` package:

Description
Displays multiple panes but allows only one to be expanded at a time.
Enables you to set the position of child nodes relative to the pane's edges. Resizing the pane does not alter the layout of the

nodes.

`BorderPane` Includes top, bottom, left, center and right area where you can place nodes. The top and bottom fill the `BorderPane`'s width and are vertically sized to their children's preferred heights. The left and right fill the height (between the top and bottom) and are horizontally sized to their children's preferred widths. The center area occupies all remaining space between the top, bottom, left and right. You might use the different areas for toolbars, navigation, a primary content area, etc.

`ButtonBar` Arranges `Button`s and other controls, typically for use in dialog boxes.

`DialogPane` Arranges a dialog box's content area, including its header, content and buttons.

`FlowPane` Lays out nodes consecutively—either horizontally or vertically. When the boundary for the pane is reached, the nodes wrap to a new line in a horizontal `FlowPane` or a new column in a vertical `FlowPane`.

`GridPane` Creates a flexible grid for laying out nodes in rows and columns.

`HBox` Arranges nodes horizontally in one row.

`Pane` The base class for layout panes. Positions nodes at exact coordinates—known as absolute positioning.

`ScrollPane` Provides scroll bars for its content, enabling users to scroll through content that cannot fit in the viewing area.

`SplitPane` Divides a layout into user-resizable areas.

`StackPane` Places nodes in a stack. Each new node is stacked atop the previous node. For example, you might use this to place text on top of images.

`Tab` A single tab in a `TabPane` layout.

`TabPane` A layout that manages multiple `Tab`s, similar to the tabs representing individual websites in a web browser window.

`TextFlow` Displays and wraps text dynamically, similar to word-processing apps like Google Docs or Microsoft Word.

`TilePane` A horizontal or vertical grid of equally sized tiles. Nodes that are tiled horizontally wrap at the `TilePane`'s width. Nodes that are tiled vertically wrap at the `TilePane`'s height.

`TitledPane` Collapsible layout with a title and content. The content can be expanded or collapsed by clicking the title.

`ToolBar` Horizontal or vertical bar containing a group of controls. Typically used to arrange `Button`s and other controls at the edges of a GUI.

`VBox` Arranges nodes vertically in one column.

Generative AI

1 Prompt genAIs with the list of layouts in the preceding table, asking them to explain typical use-cases for each.

16.3 Painter Application: RadioButtons, Mouse Events and Shapes

In this section, you'll create the following simple Painter application that allows you to drag the mouse to draw—you can compile and run each of this chapter's applications using `javac` and `java` commands similar to those you saw in Section 15.5.1:
First, we'll overview the technologies you'll use, then we'll discuss creating the application's project and building its GUI. Finally, we'll present the source code for its `Painter` and `PainterController` classes.

16.3.1 Technologies Overview

This section introduces the JavaFX features you'll use in the Painter application.

`RadioButton`s and `ToggleGroup`s

`RadioButton`s function as mutually exclusive options. You add multiple `RadioButton`s to a **`ToggleGroup`** to ensure that only one `RadioButton` in a given group is selected at a time. You'll use JavaFX Scene Builder to specify each `RadioButton`'s `ToggleGroup` in FXML; however, you can also create a `ToggleGroup` programmatically, then use a **`RadioButton`'s `setToggleGroup` method** to specify its `ToggleGroup`. In fact, some programmers create entire JavaFX GUI's programmatically without using FXML.

`BorderPane` Layout Container

A **`BorderPane` layout container** arranges controls into one or more of the five regions shown in the following diagram:
The top and bottom areas have the same width as the `BorderPane`. The left, center and right areas fill the vertical space between the top and bottom. Each area may contain only one control or one layout container—any layout container, in turn, may contain other controls, layouts and nodes. All the areas in a `BorderPane` are

optional:

- If the top or bottom area is empty, the left, center and right areas expand vertically to fill that area.
- If the left or right area is empty, the center expands horizontally to fill that area.

`TitledPane` Layout Container

A **`TitledPane` layout container** displays a title at its top and is a collapsible panel containing a layout for other nodes. You'll use `TitledPane`s to organize the application's `RadioButton`s and to help the user understand the purpose of each `RadioButton` group.

JavaFX Shapes

The **`javafx.scene.shape` package** contains various classes for 2D and 3D shape nodes that can be displayed in a scene graph. In this application, you'll programmatically create **`Circle`** objects as the user drags the mouse, then attach them to the application's drawing area to display them in the scene graph.

`Pane` Layout Container

The drawing area will be a **`Pane` layout**. Each `Circle` object will be placed at an *x*-*y* coordinate measured from the `Pane`'s upper-left corner.

Mouse Event Handling

When you drag the mouse, the application's controller responds by displaying a `Circle` in the currently selected color and pen size at the current mouse position in the `Pane`. JavaFX nodes support various mouse events, which are summarized in the following table:

MouseEvent	When event occurs for a given node
`MOUSE_CLICKED`	The user clicks a mouse button—that is, presses and releases a mouse button without moving the mouse—with the mouse cursor within that node.
`MOUSE_DRAG_ENTERED`	The mouse cursor enters a node's bounds during a mouse drag—that is, the user is moving the mouse with a mouse button pressed.
`MOUSE_DRAG_EXITED`	The mouse cursor exits the node's bounds during a mouse drag.
`MOUSE_PRESSED`	The user begins a mouse drag with the mouse cursor within that

node and continues moving the mouse with a mouse button pressed.

OnMouseDragOver A drag operation starts in a different node and continues with the mouse cursor over the given node.

OnMouseDragReleased The user completes a drag operation that began in that node.

OnMouseEntered The mouse cursor enters that node's bounds.

OnMouseExited The mouse cursor exits that node's bounds.

OnMouseMoved The mouse cursor moves within that node's bounds.

OnMousePressed The user presses a mouse button with the mouse cursor within that node's bounds.

OnMouseReleased The user releases a mouse button with the mouse cursor within that node's bounds.

For this application, you'll configure the `Pane`'s **`onMouseDragged` event handler**. JavaFX also supports other types of input events. For example, for touchscreen devices, there are touch-oriented events, and for keyboards, there are key events. For a complete list of JavaFX node events, see the `Node` class's properties that begin with the word "on" at:

Click here to view code image

```
https://openjfx.io/javadoc/23/javafx.graphics/javafx/scene/Node.html
```

Setting a Control's User Data

Each JavaFX control has a **`setUserData` method** that receives an `Object`. You can use this to store any object you'd like to associate with that control. With each drawing-color `RadioButton`, we store the specific `Color` that the `RadioButton` represents. With each pen size `RadioButton`, we store an `enum` constant for the corresponding pen size. We use these objects when handling the `RadioButton` events.

16.3.2 Creating the `Painter.fxml` File

Create a folder on your system for this example's files, then open Scene Builder and save the new FXML file as `Painter.fxml`. If you already have an FXML file open, you can also choose File > New to create a new FXML file, then save it.

16.3.3 Building the GUI

In this section, we'll discuss the Painter application's GUI. Rather than providing the exact steps as we did in Chapter 15, we'll provide general instructions for building the GUI and focus on specific details for new concepts. As you build the GUI, keep in mind that it's often easier to manipulate layouts and controls via Scene Builder's Hierarchy section than directly in the stage design area.

fx:id Property Values for This Application's Controls

The following diagram shows the fx:id properties of the Painter application's programmatically manipulated controls. As you build the GUI, you should set the corresponding fx:id properties in the FXML document, as we discussed in Chapter 15:

Step 1: Adding a `BorderPane` as the Root Layout Node

Drag a `BorderPane` from the Scene Builder Library window's Containers section onto the content panel.

Step 2: Configuring the `BorderPane`

Set the `BorderPane`'s Pref Width and Pref Height properties to `640` and `480`. Recall that the stage's size is determined based on the size of the root node in the FXML document. For the `BorderPane`, set the Padding property's four values to `8` to inset it from the stage's edges.

Step 3: Adding the `Pane` and `VBox`

Drag a `Pane` into the center area and a `VBox` into the `BorderPane`'s left area. As you drag over the `BorderPane`, Scene Builder shows the layout's five regions and highlights the one where the item you're dragging will appear when you release the mouse. Set the `Pane`'s fx:id to `drawingAreaPane`.

For the `VBox`, set its Spacing property (in the Inspector's Layout section) to `8` to add some vertical space between the controls that you'll place in this container. Set its right Margin property to `8` to add some horizontal spacing between the `VBox` and the `Pane`. Also, set its Max Height property to `MAX_VALUE`. This will enable the `VBox` to be as wide as it needs to be to accommodate its child nodes and occupy the full column height.

Reset the `Pane`'s Pref Width and Pref Height to their default `USE_COMPUTED_SIZE` values, and set its Max Width and Max Height to

MAX_VALUE so that it occupies the entire width and height of the BorderPane's center area. In the Pane's JavaFX CSS category of the Inspector window's Properties section, click the field below Style (initially empty) and select

```
-fx-background-color
```

so you can specify the Pane's background color. In the field to its right, specify white.

Step 4: Adding the TitledPanes to the VBox

From the Library window's Containers section, drag two TitledPane (empty) objects onto the VBox. For the first TitledPane, set its Text property to Drawing Color. For the second, set its Text property to Pen Size.

Step 5: Customizing the TitledPanes

Each TitledPane in the completed GUI contains multiple RadioButtons. We'll use a VBox in each TitledPane to help arrange those controls. Drag a VBox onto each TitledPane. Set each VBox's Spacing property to 8 and Pref Width and Pref Height to USE_COMPUTED_SIZE so the VBoxes will be sized based on their contents.

Step 6: Adding the RadioButtons to the VBox

From the Library window's Controls section, drag four RadioButtons onto the VBox for the Drawing Color TitledPane and three RadioButtons onto the VBox for the Pen Size Titled-Pane. Configure their Text properties and fx:ids as shown previously. Next, select the blackRadioButton and ensure its Selected property is checked, then do the same for the mediumRadioButton.

Step 7: Specifying the ToggleGroups for the RadioButtons

Select all four RadioButtons in the first TitledPane's VBox. In the Inspector's Properties section, set the Toggle Group property to colorToggleGroup. When the application loads the FXML file at execution time, the FXMLLoader will create a ToggleGroup object by that name and associate these four RadioButtons with it to ensure that only one is selected at a time. Repeat this step for the three RadioButtons in the second TitledPane's VBox, but set the Toggle

Group property to `sizeToggleGroup`.

Step 8: Changing the `TitledPane`s' Preferred Width and Height

For each `TitledPane`, set its Pref Width and Pref Height to `USE_COMPUTED_SIZE` so the `TitledPane`s will be sized based on their contents.

Step 9: Adding the `Button`s

Add two `Button`s below the `TitledPane`s, then configure their Text properties and fx:ids as shown previously. Set each `Button`'s Max Width property to `MAX_VALUE` so that they fill the `VBox`'s width.

Step 10: Setting the `VBox`'s Width

We want the `VBox` to be only as wide as required for the controls in that column. Select the `VBox` in the Document window's Hierarchy section. Set its Min Width and Pref Width properties to `USE_COMPUTED_SIZE`, then set the Max Width to `USE_PREF_SIZE` to indicate that the maximum width should be the preferred width. Also, reset the Max Height to its default `USE_COMPUTED_SIZE` value. The GUI should now appear as shown at the beginning of this section.

Step 11: Specifying the Controller Class's Name

As we mentioned in Section 15.5.2, in a JavaFX FXML application, the application's controller class typically defines instance variables for interacting with controls programmatically, as well as event-handling methods. To ensure that an object of the controller class is created when the application loads the FXML file at runtime, specify the controller class's name in the FXML file:

- Expand Scene Builder's Controller window (located below the Hierarchy window).
- In the Controller Class field, type `PainterController`.

Step 12: Specifying the Event-Handler Method Names

Next, specify in the Inspector window's Code section the names of the methods that will be called to handle specific events:

- For the `drawingAreaPane`, go to the Code section's Mouse heading and specify `drawingAreaMouseDragged` as the On Mouse Dragged event handler. This method will draw a circle in the specified color and size for each mouse-dragged event.
- For the four Drawing Color `RadioButton`s, specify `colorRadioButtonSelected` as each `RadioButton`'s On Action event handler. This method will set the current drawing color based on the user's selection.
- For the three Pen Size `RadioButton`s, specify `sizeRadioButtonSelected` as each `RadioButton`'s On Action event handler. This method will set the current pen size based on the user's selection.
- For the Undo `Button`, specify `undoButtonPressed` as the On Action event handler. This method will remove the last circle the user drew on the screen.
- For the Clear `Button`, specify `clearButtonPressed` as the On Action event handler. This method will clear the entire drawing.

Step 13: Generating a Sample Controller Class

As you saw in Section 15.5, Scene Builder generates the initial controller-class skeleton when you select View > Show Sample Controller Skeleton. You can copy this code into a `PainterController.java` file and store the file in the same folder as `Painter.fxml`. We show the completed `PainterController` class in Section 16.3.5.

16.3.4 `Painter` Subclass of `Application`

Figure 16.1 shows the `Painter` subclass of `Application` that launches the application. This class performs the same tasks described for the Tip Calculator application in Section 15.5.4.

Click here to view code image

```
1  // Fig. 16.1: Painter.java
2  // Class that loads and displays the Painter's GUI.
3  import javafx.application.Application;
4  import javafx.fxml.FXMLLoader;
5  import javafx.scene.Parent;
```

```
import javafx.scene.Scene;
import javafx.stage.Stage;

public class Painter extends Application {
   @Override
   public void start(Stage stage) throws Exception {
      Parent root =
         FXMLLoader.load(getClass().getResource("Painter.fxml"));

      Scene scene = new Scene(root);
      stage.setTitle("Painter"); // displayed in window's title bar
      stage.setScene(scene);
      stage.show();
   }

   public static void main(String[] args) {
      launch(args);
   }
}
```

Fig. 16.1 | Class that loads and displays the Painter's GUI.

16.3.5 `PainterController` Class

Figure 16.2 shows the final version of the `PainterController` class, which we broke into parts for discussion purposes. Recall from Chapter 15 that the controller class defines instance variables for interacting with controls programmatically, as well as event-handling methods. The controller class may also declare additional instance variables, `static` variables and methods that support the application's operation.

Click here to view code image

```
// Fig. 16.2: PainterController.java
// Controller for the Painter application
import javafx.event.ActionEvent;
import javafx.fxml.FXML;
import javafx.scene.control.RadioButton;
import javafx.scene.control.ToggleGroup;
import javafx.scene.input.MouseEvent;
import javafx.scene.layout.Pane;
import javafx.scene.paint.Color;
import javafx.scene.paint.Paint;
import javafx.scene.shape.Circle

public class PainterController {
```

Fig. 16.2 | Controller for the Painter application.

PenSize enum

Lines 15–25 define the nested `enum` type `PenSize`, which specifies three pen sizes—`SMALL`, `MEDIUM` and `LARGE`. Each contains a radius that will be used when creating a `Circle` object to display in response to a mouse-drag event. Java allows you to declare `class`es, `record`s, `interface`s and `enum`s as **nested types** inside other classes. Most previously presented types were top-level, not nested in other types. The `PenSize enum` is declared as a `private` nested type because it's used only by the `PainterController`.

Click here to view code image

```
14      // enum representing pen sizes
15      private enum PenSize {
16         SMALL(2),
17         MEDIUM(4),
18         LARGE(6);
19
20         private final int radius;
21
22         PenSize(int radius) {this.radius = radius;} // constructor
23
24         public int getRadius() {return radius;}
25      };
26
```

Instance Variables

Lines 28–37 declare the `@FXML` instance variables the controller uses to programmatically interact with the GUI. Recall that the names of these variables must match the fx:id values in `Painter.fxml`; otherwise, the `FXMLLoader` will not be able to connect the GUI components to the instance variables. Two of the `@FXML` instance variables are `ToggleGroup`s— the `RadioButton` event handlers use these to determine which `RadioButton` was selected. Lines 40–41 define instance variables to store the current `PenSize` and drawing `Color`.

Click here to view code image

```
27      // instance variables that refer to GUI components
28      @FXML private RadioButton blackRadioButton;
29      @FXML private RadioButton redRadioButton;
30      @FXML private RadioButton greenRadioButton;
```

```
31      @FXML private RadioButton blueRadioButton;
32      @FXML private RadioButton smallRadioButton;
33      @FXML private RadioButton mediumRadioButton;
34      @FXML private RadioButton largeRadioButton;
35      @FXML private Pane drawingAreaPane;
36      @FXML private ToggleGroup colorToggleGroup;
37      @FXML private ToggleGroup sizeToggleGroup;
38
39      // instance variables for managing Painter state
40      private PenSize radius = PenSize.MEDIUM; // radius of circle
41      private Paint brushColor = Color.BLACK; // drawing color
42
```

Method `initialize`

Recall that when the `FXMLLoader` creates a controller-class object, it determines whether the controller class contains an `initialize` method declared with `@FXML` and no parameters and, if so, calls that method to initialize the controller. Lines 44–54 define method `initialize` to specify each `RadioButton`'s corresponding user data object—a `Color` or a `PenSize`. We use these objects in the `RadioButton`s' event handlers.

Click here to view code image

```
43      // set user data for the RadioButtons
44      @FXML
45      private void initialize() {
46         // user data on a control can be any Object
47         blackRadioButton.setUserData(Color.BLACK);
48         redRadioButton.setUserData(Color.RED);
49         greenRadioButton.setUserData(Color.GREEN);
50         blueRadioButton.setUserData(Color.BLUE);
51         smallRadioButton.setUserData(PenSize.SMALL);
52         mediumRadioButton.setUserData(PenSize.MEDIUM);
53         largeRadioButton.setUserData(PenSize.LARGE);
54      }
55
```

`drawingAreaMouseDragged` Event Handler

Lines 57–62 define `drawingAreaMouseDragged`, which responds to drag events in the `drawingAreaPane`. Each mouse event handler receives a **`MouseEvent`** parameter (package **`javafx.scene.input`**). When the event occurs, this parameter contains information about the event, such as its location, whether any mouse buttons were pressed, which node the user interacted with and more. You

specified `drawingAreaMouseDragged` in Scene Builder as the `drawingAreaPane`'s On Mouse Dragged event handler.

Click here to view code image

```
56      // handles drawingArea's onMouseDragged MouseEvent
57      @FXML
58      private void drawingAreaMouseDragged(MouseEvent e) {
59         var newCircle = new Circle(e.getX(), e.getY(),
60            radius.getRadius(), brushColor);
61         drawingAreaPane.getChildren().add(newCircle);
62      }
63
```

Lines 59–60 create a new `Circle` object using its constructor that receives its center's *x*-*y* coordinates, its radius and its `Color`. Next, line 61 attaches the new `Circle` to the `drawingAreaPane`. A layout's **`getChildren` method** returns an **`ObservableList<Node>`** collection containing its child nodes. An `ObservableList` has methods for adding and removing elements. Line 61 uses the `ObservableList`'s `add` method to add a new `Node` to the `drawingAreaPane`—all JavaFX shapes inherit indirectly from class `Node` in the `javafx.scene` package. You'll learn more about `ObservableList` later in this chapter.

`colorRadioButtonSelected` Event Handler

Method `colorRadioButtonSelected` (lines 65–70) responds to the `ActionEvent`s of the Drawing Color `RadioButton`s, which occur when a new color `RadioButton` is selected. You specified this method in Scene Builder as the On Action event handler for all four Drawing Color `RadioButton`s. Lines 68–69 set the current drawing `Color`. `ColorToggleGroup`'s **`getSelectedToggle` method** returns the selected `Toggle`. `RadioButton` is one of several controls (others are `RadioButtonMenuItem` and `ToggleButton`) that implement interface **`Toggle`**. The **`Toggle`'s `getUserData` method** returns the user data `Object` we associated with the selected `RadioButton` in method `initialize`—for `ColorToggleGroup`'s `RadioButton`s, the `Object` is always a `Color`, so we cast the `Object` to a `Color` and assign it to `brushColor`.

Click here to view code image

```
64      // handles color RadioButton's ActionEvents
65      @FXML
66      private void colorRadioButtonSelected(ActionEvent e) {
```

```
67          // user data for each color RadioButton is the corresponding Color
68          brushColor =
69             (Color) colorToggleGroup.getSelectedToggle().getUserData();
70       }
71
```

`sizeRadioButtonSelected` Event Handler

Lines 73–78 define `sizeRadioButtonSelected`, which responds to the Pen Size `RadioButton`s' `ActionEvent`s. You specified this method as the On Action event handler for all three Pen Size `RadioButton`s. Lines 76–77 set the current `PenSize`, using the same approach as setting the current color in `colorRadioButtonSelected`. In this case, the user data `Object` is always a `PenSize`.

Click here to view code image

```
72       // handles size RadioButton's ActionEvents
73       @FXML
74       private void sizeRadioButtonSelected(ActionEvent e) {
75          // user data for each size RadioButton is the corresponding PenSize
76          radius =
77             (PenSize) sizeToggleGroup.getSelectedToggle().getUserData();
78       }
79
```

`undoButtonPressed` Event Handler

Lines 81–89 define `undoButtonPressed`, which responds to an `ActionEvent` from the `undoButton` by removing the last `Circle` displayed. You specified this method in Scene Builder as the `undoButton`'s On Action event handler. To undo the last `Circle`, we remove the last child from the `drawingAreaPane`'s collection of child nodes. First, line 83 gets the number of elements in that collection. Then, if that's greater than 0, line 87 removes the node at the last index in the collection.

Click here to view code image

```
80       // handles Undo Button's ActionEvents
81       @FXML
82       private void undoButtonPressed(ActionEvent event) {
83          int count = drawingAreaPane.getChildren().size();
84
85          // if there are any shapes, remove the last one added
86          if (count > 0) {
87             drawingAreaPane.getChildren().remove(count - 1);
```

```
88          }
89       }
90
```

`clearButtonPressed` Event Handler

Lines 92–95 define `clearButtonPressed`, which responds to the `ActionEvent` from the `clearButton` by clearing `drawingAreaPane`'s collection of child nodes. You specified this event handler in Scene Builder as the `clearButton`'s On Action event handler. Line 94 clears the collection of child nodes to erase the entire drawing.

Click here to view code image

```
91       // handles Clear Button's ActionEvents
92       @FXML
93       private void clearButtonPressed(ActionEvent event) {
94          drawingAreaPane.getChildren().clear(); // clear the canvas
95       }
96    }
```

Generative AI

1 Prompt genAIs, asking them to explain best practices for using `RadioButton`s in a JavaFX application.

2 Prompt genAIs, asking them to provide examples of a `Node`'s mouse events and how they're typically used.

16.4 Color Chooser Application: Property Bindings and Property Listeners

This section demonstrates property bindings and property listeners by building a Color Chooser application. The following diagram shows the result of using the Red and Green `Slider`s to create an opaque orange color:
The following diagram shows the result of using the Red, Green and Alpha `Slider`s to create a semitransparent orange color—notice that the semitransparent orange mixes with the color of the circle behind the colored square:

16.4.1 Technologies Overview

Let's introduce the technologies you'll use to build the Color Chooser.

RGBA Colors

The application uses the **RGBA color model** to display a rectangle of color based on the values of four `Slider`s. RGBA represents every color by its red, green and blue color values, each ranging from 0 to 255—0 denotes no color and 255 full color. For example, a color with a red value of 0 would contain no red component. The alpha value (A) ranges from 0.0 to 1.0 and represents a color's opacity—0.0 is transparent, and 1.0 is opaque. The two colors in the previous sample outputs have the same RGB values, but the color displayed in Part (b) is semitransparent. You'll create an RGBA `Color` object then use it to fill a `Rectangle` with that `Color`.

Properties of a Class

JavaFX uses properties extensively. A **property** is an instance of a class that implements JavaFX's **`Property` interface** (**package `javafx.beans.property`**) and is accessed via *set* and *get* methods. In general, the pair of methods that define a read/write property have the form:

Click here to view code image

```
public final void setPropertyName(Type propertyName)
public final Type getPropertyName()
```

Typically, such methods manipulate a corresponding `private` instance variable with the same name as the property, but this is not required. For example, in a `Time` class, methods `setHour` and `getHour` together represent an `hour` property—they'd typically manipulate a private `hour` instance variable. If the property represents a `boolean` value, its *get* method name would begin with "`is`" rather than "`get`"—for example, `ArrayList` method `isEmpty`. Methods that define properties should be declared `final` to prevent subclasses from overriding the methods, which could lead to unexpected results in client code.

Property Bindings

JavaFX properties are implemented in a manner that makes them observable—when a property's value changes, other objects can respond accordingly. This is similar to event handling. One way to

respond to a property change is via a **property binding**, which enables a property of one object to be updated when a property of another object changes. You'll use property bindings to update a `TextField` with the corresponding `Slider`'s current value when the user moves that `Slider`'s thumb. Property bindings are not limited to JavaFX controls. Package **`javafx.beans.property`** contains many classes that you can use to define bindable properties in your own classes.

Property Listeners

Property listeners are similar to property bindings. A **property listener** is an event handler that's invoked when a property's value changes. In the event handler, you can respond to the property change in a manner appropriate for your application. In this application, when a `Slider`'s value changes, a property listener will store the value in a corresponding instance variable, create a new `Color` based on the values of all four `Slider`s and set that `Color` as the fill color of a `Rectangle` object that displays the current color. For more information on properties, property bindings and property listeners, visit:

Click here to view code image

```
https://deitel.com/JavaFXPropertyBindingTutorial
```

16.4.2 Building the GUI

In this section, we'll discuss the Color Chooser application's GUI. Rather than providing the exact steps as we did in Chapter 15, we'll provide general instructions for building the GUI and focus on specific details for new concepts. As you build the GUI, recall that it's often easier to manipulate layouts and controls via the Scene Builder Document window's Hierarchy section than directly in the stage design area. Before proceeding, open Scene Builder and create an FXML file named `ColorChooser.fxml`.

fx:id Property Values for This Application's Controls

The following diagram shows the fx:id properties for the Color Chooser application's pro-grammatically manipulated controls. You should set these controls' fx:id properties in the FXML document, as you learned in Chapter 15.

Step 1: Adding a `GridPane`

Drag a `GridPane` from the Library window's Containers section onto Scene Builder's content panel.

Step 2: Configuring the `GridPane`

This application's `GridPane` requires four rows and four columns. Use the techniques you've learned previously to add two columns and one row to the `GridPane`. Set the `GridPane`'s Hgap and Padding properties to `8` to inset the `GridPane` from the stage's edges and provide space between its columns.

Step 3: Adding the Controls

Using the previous diagram as a guide, add the `Label`s, `Slider`s, `TextField`s, a `Circle` and a `Rectangle` to the `GridPane`—`Circle` and `Rectangle` are located in the Scene Builder Library's Shapes section. When adding the `Circle` and `Rectangle`, place both in the rightmost column's first row, adding the `Circle` before the `Rectangle` so it will be behind the rectangle in the layout. Set the text of the `Label`s and `TextField`s as shown and set all the appropriate fx:id properties as you add each control.

Step 4: Configuring the `Slider`s

For the red, green and blue `Slider`s, set the Max properties to `255` (the maximum amount of a given color in the RGBA color scheme). For the alpha `Slider`, set its Max property to `1.0` (the maximum opacity in the RGBA color scheme).

Step 5: Configuring the `TextField`s

Set all of the `TextField`'s Pref Width properties to `50`.

Step 6: Configuring the `Rectangle`

Set the `Rectangle`'s Width and Height properties to `100`, then set its Row Span property to `Remaining` so it spans all four rows in that column.

Step 7: Configuring the `Circle`

Set the `Circle`'s Radius property to `40`, then set its Row Span property to `Remainder` so it spans all four rows in that column.

Step 8: Configuring the Rows

Set all four rows' Pref Height properties to `USE_COMPUTED_SIZE` so the rows are only as tall as their content.

Step 9: Configuring the Columns

Set all four columns' Pref Width properties to `USE_COMPUTED_SIZE` so the columns are only as wide as their content. For the leftmost column, set the Halignment property to `RIGHT`. For the rightmost column, set the Halignment property to `CENTER`.

Step 10: Configuring the `GridPane`

Set the `GridPane`'s Pref Width and Pref Height properties to `USE_COMPUTED_SIZE` so it sizes itself based on its contents. Your GUI is now complete.

Step 11: Specifying the Controller Class's Name

To ensure that an object of the controller class is created when the application loads the FXML file at runtime, specify `ColorChooserController` as the controller class's name in the FXML file.

Step 12: Generating a Sample Controller Class

Select View > Show Sample Controller Skeleton, then copy this code into a `ColorChooserController.java` file and store the file in the same folder as `ColorChooser.fxml`. We show the completed `ColorChooserController` class in Section 16.4.4.

16.4.3 `ColorChooser` Subclass of `Application`

Figure 16.3 shows the `ColorChooser` subclass of `Application`. This class loads the FXML and displays the application as in the prior JavaFX examples.

Click here to view code image

```
1  // Fig. 16.3: ColorChooser.java
2  // Class that loads and displays the ColorChooser's GUI.
3  import javafx.application.Application;
4  import javafx.fxml.FXMLLoader;
5  import javafx.scene.Parent;
```

```
6  import javafx.scene.Scene;
7  import javafx.stage.Stage;
8
9  public class ColorChooser extends Application {
10    @Override
11    public void start(Stage stage) throws Exception {
12       Parent root =
13          FXMLLoader.load(getClass().getResource("ColorChooser.fxml"));
14
15       var scene = new Scene(root);
16       stage.setTitle("Color Chooser");
17       stage.setScene(scene);
18       stage.show();
19    }
20
21    public static void main(String[] args) {
22       launch(args);
23    }
24 }
```

Fig. 16.3 | Class that loads and displays the Color Chooser's GUI.

16.4.4 `ColorChooserController` Class

Figure 16.4 shows the final version of class `ColorChooserController`. We broke the code into parts for discussion purposes.

Click here to view code image

```
1  // Fig. 16.4: ColorChooserController.java
2  // Controller for the ColorChooser application
3  import javafx.beans.value.ChangeListener;
4  import javafx.beans.value.ObservableValue;
5  import javafx.fxml.FXML;
6  import javafx.scene.control.Slider;
7  import javafx.scene.control.TextField;
8  import javafx.scene.paint.Color;
9  import javafx.scene.shape.Rectangle;
10
11 public class ColorChooserController {
```

Fig. 16.4 | Controller for the `ColorChooser` application.

Instance Variables

Lines 13–27 declare the controller's instance variables. The

variables `red`, `green`, `blue` and `alpha` store the `redSlider`, `greenSlider`, `blueSlider` and `alphaSlider` current values. We use these to update the `colorRectangle`'s fill color when the user adjusts a `Slider`.

Click here to view code image

```
12      // instance variables for interacting with GUI components
13      @FXML private Slider redSlider;
14      @FXML private Slider greenSlider;
15      @FXML private Slider blueSlider;
16      @FXML private Slider alphaSlider;
17      @FXML private TextField redTextField;
18      @FXML private TextField greenTextField;
19      @FXML private TextField blueTextField;
20      @FXML private TextField alphaTextField;
21      @FXML private Rectangle colorRectangle;
22
23      // instance variables for managing
24      private int red = 0;
25      private int green = 0;
26      private int blue = 0;
27      private double alpha = 1.0;
28
```

Method `initialize`

Lines 29–66 define method `initialize` to complete the controller's initialization after the `FXMLLoader` creates the GUI. In this application, `initialize` configures the property bindings and property listeners.

Click here to view code image

```
29      @FXML
30      private void initialize() {
31         // bind TextField values to corresponding Slider values
32         redTextField.textProperty().bind(
33            redSlider.valueProperty().asString("%.0f"));
34         greenTextField.textProperty().bind(
35            greenSlider.valueProperty().asString("%.0f"));
36         blueTextField.textProperty().bind(
37            blueSlider.valueProperty().asString("%.0f"));
38         alphaTextField.textProperty().bind(
39            alphaSlider.valueProperty().asString("%.2f"));
40
41         // listeners that set Rectangle's fill based on Slider changes
42         redSlider.valueProperty().addListener(
```

```
43              (ov, oldValue, newValue) -> {
44                 red = newValue.intValue();
45                 colorRectangle.setFill(Color.rgb(red, green, blue, alpha));
46              }
47           );
48           greenSlider.valueProperty().addListener(
49              (ov, oldValue, newValue) -> {
50                 green = newValue.intValue();
51                 colorRectangle.setFill(Color.rgb(red, green, blue, alpha));
52              }
53           );
54           blueSlider.valueProperty().addListener(
55              (ov, oldValue, newValue) -> {
56                 blue = newValue.intValue();
57                 colorRectangle.setFill(Color.rgb(red, green, blue, alpha));
58              }
59           );
60           alphaSlider.valueProperty().addListener(
61              (ov, oldValue, newValue) -> {
62                 alpha = newValue.doubleValue();
63                 colorRectangle.setFill(Color.rgb(red, green, blue, alpha));
64              }
65           );
66        }
67     }
```

Property-to-Property Bindings

Lines 32–39 set up property bindings between each `Slider`'s value and the corresponding `TextField`'s text so `Slider` changes update the corresponding `TextField`. Consider lines 32–33, which bind the `redSlider`'s `valueProperty` to the `redTextField`'s `textProperty`:

Click here to view code image

```
redTextField.textProperty().bind(
   redSlider.valueProperty().asString("%.0f"));
```

A `TextField`'s **`textProperty` method** returns its `text` property as a **`StringProperty`** (package `javafx.beans.property`). **`StringProperty` method `bind`** receives an **`ObservableValue`** as an argument. When its `ObservableValue` changes, the bound property updates. In this case, the `ObservableValue` is returned by

Click here to view code image

```
redSlider.valueProperty().asString("%.0f")
```

The `Slider`'s `valueProperty` method returns its `value` property as a

`DoubleProperty`, which is an observable `double` value. However, the `TextField`'s `text` property must be bound to an observable `String`. So, we call the `DoubleProperty` **method** `asString` to get a `StringBinding` object—an `ObservableValue` that provides a `String` representation of the `DoubleProperty`. This version of `asString` receives a format-control `String` specifying the `DoubleProperty`'s `String` format.

Once you bind a `TextField`'s `textProperty` to a property of another control, the user may not edit the `TextField`, even though we did not disable its editable property in Scene Builder. In an end-of-chapter genAI exercise, you'll investigate bidirectional property bindings, which would allow the `TextField` to remain editable by the user.

Property Listeners

You can register a property listener if you need to perform an arbitrary task when a property's value changes. Lines 42–65 register property listeners for the `Slider`s' `value` properties. Consider lines 42–47, which register the `ChangeListener` that handles changes to the `redSlider`'s value. As in Section 15.5, we implement the `ChangeListener` interface with a lambda. Each `ChangeListener` stores the `newValue` parameter's `int` value in the corresponding `ColorChooserController` instance variable, then calls the `colorRectangle`'s `setFill` method to change its color, using `Color` method `rgb` to create the new `Color` object.

Generative AI

1 Prompt genAIs, asking them to discuss the difference between JavaFX property listeners and property bindings.

2 Prompt genAIs for a list of commonly used property bindings in JavaFX.

16.5 Cover Viewer Application: Data-Driven GUIs with JavaFX Collections

Often, an application needs to edit and display data. JavaFX provides a comprehensive model for allowing GUIs to interact with data. In this section, you'll build the Cover Viewer application (shown in the following diagram), which binds a list of `Book` objects

to a `ListView`. When the user selects an item in the `ListView`, the corresponding `Book`'s cover image is displayed in an `ImageView`.

16.5.1 Technologies Overview

This application uses a `ListView` control to display a list of book titles. Though you can individually add items to a `ListView`, this application binds an **`ObservableList`** object to the `ListView`. If you change an `ObservableList`'s elements, its observer (the `ListView`) will automatically be notified. Package **`javafx.collections`** defines `ObservableList` (which, similar to an `ArrayList`, implements the `List` interface) and other observable collection interfaces. The package also contains class **`FXCollections`**, which provides `static` methods for creating and manipulating observable collections. You'll use a property listener to display the correct image when the user selects an item from the `ListView`. For this control, the property that changes is the selected item.

Before building the application, copy the `images` folder (which contains the `large` and `small` subfolders) from this chapter's examples folder into the folder where you'll save this application's FXML file and the `CoverViewer.java` and `CoverViewerController.java` source-code files. Though you'll use only the `large` images in this example, you'll copy this application's folder to create Section 16.6's example, which uses both sets of images.

16.5.2 Building the GUI

In this section, we'll discuss the Cover Viewer application's GUI. Use Scene Builder to create a new FXML file and save it as `CoverViewer.fxml`.

fx:id Property Values for This Application's Controls

The following diagram shows the fx:id properties of the Cover Viewer application's pro-grammatically manipulated controls. As you build the GUI, set the corresponding fx:id properties in the FXML document.

Adding and Configuring the Controls

Using the techniques you learned previously, create a `BorderPane`. In the left area, place a `ListView` control, and in the center area,

place an `ImageView` control. For the `ListView`, set the following properties:

- Margin—`8` (for the right margin) to separate the `ListView` from the `ImageView`
- Pref Width—`200`
- Max Height—`MAX_VALUE`
- Min Width, Min Height, Pref Height and Max Width—`USE_COMPUTED_SIZE`

For the `ImageView`, set the Fit Width and Fit Height properties to `370` and `480`. To size the `BorderPane` based on its contents, set its Pref Width and Pref Height to `USE_COMPUTED_SIZE`. Also, set its Padding property to `8` to inset the `BorderPane` from the stage.

Specifying the Controller Class's Name

To ensure that an object of the controller class is created when the application loads the FXML file at runtime, specify `CoverViewerController` as the controller class's name in the FXML file, as you've done previously.

Generating a Sample Controller Class

Select View > Show Sample Controller Skeleton, then copy this code into a `CoverViewerController.java` file and store the file in the same folder as `CoverViewer.fxml`. We show the completed `CoverViewerController` class in Section 16.5.4.

16.5.3 CoverViewer Subclass of `Application`

Figure 16.5 shows the `CoverViewer` subclass of `Application`.

Click here to view code image

```
1  // Fig. 16.5: CoverViewer.java
2  // Class that loads and displays the CoverViewer's GUI.
3  import javafx.application.Application;
4  import javafx.fxml.FXMLLoader;
5  import javafx.scene.Parent;
6  import javafx.scene.Scene;
7  import javafx.stage.Stage;
8
9  public class CoverViewer extends Application {
```

```
10      @Override
11      public void start(Stage stage) throws Exception {
12         Parent root =
13            FXMLLoader.load(getClass().getResource("CoverViewer.fxml"));
14
15         var scene = new Scene(root);
16         stage.setTitle("Cover Viewer");
17         stage.setScene(scene);
18         stage.show();
19      }
20
21      public static void main(String[] args) {
22         launch(args);
23      }
24   }
```

Fig. 16.5 | Class that loads and displays the Cover Viewer's GUI.

16.5.4 `CoverViewerController` Class

Figure 16.6 shows the completed `CoverViewerController` class, which we've broken into parts for discussion purposes.

Click here to view code image

```
1   // Fig. 16.6: CoverViewerController.java
2   // Controller for Cover Viewer application
3   import javafx.beans.value.ObservableValue;
4   import javafx.collections.FXCollections;
5   import javafx.collections.ObservableList;
6   import javafx.fxml.FXML;
7   import javafx.scene.control.ListView;
8   import javafx.scene.image.Image;
9   import javafx.scene.image.ImageView;
10
11  public class CoverViewerController {
```

Fig. 16.6 | Controller for Cover Viewer application.

`@FXML` Instance Variables

Lines 13–14 declare the controller's `@FXML` instance variables. Note that `ListView` is a generic class. In this case, the `ListView` displays `Book` objects. We implemented `Book` as a `record` class containing three `String` instance variables:

- `title`—the book's title.

- `thumbImage`—the path to the book's thumbnail image (used in Section 16.6's example).
- `largeImage`—the path to the book's large cover image.

The `record` class also provides a custom `toString` method that returns the `Book`'s title. You should copy class `Book` from this chapter's examples folder into the folder that contains `CoverViewer.fxml`, `CoverViewer.java` and `CoverViewerController.java`.

Click here to view code image

```
12      // instance variables for interacting with GUI
13      @FXML private ListView<Book> booksListView;
14      @FXML private ImageView coverImageView;
15
```

Instance Variable `books`

Lines 17–18 define the `books` instance variable as an `ObservableList<Book>` and initialize it by calling **`FXCollections` `static` method `observableArrayList`**. This method returns an empty collection object that implements the `ObservableList` interface and has capabilities similar to an `ArrayList`.

Click here to view code image

```
16      // stores the list of Book Objects
17      private final ObservableList<Book> books =
18         FXCollections.observableArrayList();
19
```

Initializing the `books ObservableList`

Lines 23–38 in the `initialize` method add `Book` objects to the `books` collection. Line 39 passes this collection to **`ListView` method `setItems`**, which binds the `ListView` to the `ObservableList`. This data binding allows the `ListView` to display the `Book` objects automatically. By default, the `ListView` displays each `Book`'s `String` representation. You'll customize this in Section 16.6.

Click here to view code image

```
20      // initialize controller
21      public void initialize() {
```

```
22          // populate the ObservableList<Book>
23          books.add(new Book("C How to Program",
24             "/images/small/chtp.jpg", "/images/large/chtp.jpg"));
25          books.add(new Book("C++ How to Program",
26             "/images/small/cpphtp.jpg", "/images/large/cpphtp.jpg"));
27          books.add(new Book("Intro to Python",
28             "/images/small/python.jpg", "/images/large/python.jpg"));
29          books.add(new Book("Internet and World Wide Web How to Program",
30             "/images/small/iw3htp.jpg", "/images/large/iw3htp.jpg"));
31          books.add(new Book("Java How to Program",
32             "/images/small/jhtp.jpg", "/images/large/jhtp.jpg"));
33          books.add(new Book("Python for Programmers",
34             "/images/small/pythonfp.jpg", "/images/large/pythonfp.jpg"));
35          books.add(new Book("Visual Basic How to Program",
36             "/images/small/vbhtp.jpg", "/images/large/vbhtp.jpg"));
37          books.add(new Book("Visual C# How to Program",
38             "/images/small/vcshtp.jpg", "/images/large/vcshtp.jpg"));
39          booksListView.setItems(books); // bind booksListView to books
40
41          // when ListView selection changes, show large cover in ImageView
42          booksListView.getSelectionModel().selectedItemProperty().
43             addListener(
44                (ov, oldValue, newValue) -> {
45                   coverImageView.setImage(
46                      new Image(newValue.largeImage()));
47                }
48             );
49       }
50    }
```

Listening for `ListView` Selection Changes

To synchronize the displayed book cover with the currently selected book, we listen for changes to the `ListView`'s selected item. By default, a `ListView` supports single selection—one item at a time may be selected. `ListView`s also support multiple selection. The `ListView`'s **`MultipleSelectionModel`** (a subclass of **`SelectionModel`** from package `javafx.scene.control`) manages the selection type. This object contains observable properties and various methods for manipulating the corresponding `ListView`'s items.

To respond to selection changes, you register a listener for the `MultipleSelection-Model`'s `selectedItem` property (lines 42–48). **`ListView` method `getSelectionModel`** returns a `MultipleSelectionModel` object. In this example, **`MultipleSelectionModel`'s `selectedItemProperty` method** returns a **`ReadOnlyObjectProperty<Book>`**, and the corresponding `ChangeListener` receives as its `oldValue` and `newValue` parameters

the previously selected and newly selected `Book` objects, respectively.

Lines 45–46 use `newValue`'s large image path to initialize a new **`Image`** (package `javafx.scene.image`)—this loads the image from that path. We then pass the new `Image` to the `coverImageView`'s **`setImage` method** to display the `Image`.

Generative AI

1 Prompt genAIs for an overview of the JavaFX collections in `javafx.collections` package and typical uses of each.

2 Prompt genAIs for an overview of commonly used methods provided by JavaFX class `FXCollections`.

16.6 Cover Viewer Application: Customizing `ListView` Cells

In the preceding example, the `ListView` displayed a `Book`'s `String` representation (its title). In this example, you'll create a custom `ListView` cell factory to create cells that display each book as its thumbnail image and title using a `VBox`, an `ImageView` and a `Label`, as shown at the left side of the following screen capture:

16.6.1 Technologies Overview

`ListCell` Generic Class for Custom `ListView` Cell Formats

You saw in Section 16.5 that `ListView` cells display `ListView` items' `String` representations by default. To create a custom cell format, you first define a subclass of the **`ListCell` generic class** (package `javafx.scene.control`) that specifies how to create a `ListView` cell. As the `ListView` displays items, it gets `ListCell`s from its cell factory. You'll use **`ListView`'s `setCellFactory` method** to replace the default cell factory with one that returns objects of your `ListCell` subclass. You'll override this class's **`updateItem` method** to specify the cells' custom layout and contents.

Programmatically Creating Layouts and Controls

So far, you've created GUIs visually using JavaFX Scene Builder. In this application, you'll also create a portion of the GUI programmatically. In fact, everything we've shown you in Scene Builder can also be accomplished directly in Java code. For this application, you'll create and configure a custom `ListView` cell format using a `VBox` layout containing an `ImageView` and a `Label`.

16.6.2 Copying the `CoverViewer` Application

This application's FXML layout and classes `Book` and `CoverViewer` are identical to those in Section 16.5, and the `CoverViewerController` class has only one new statement. For this example, we'll show a new class that implements the custom `ListView` cell factory and `CoverViewerController`'s new statement. Rather than creating a new application from scratch, copy the `CoverViewer` application from the previous example into a new folder named `CoverViewerCustomListView`.

16.6.3 `ImageTextCell` Custom `ListView` Cell Layout

Class `ImageTextCell` (Fig. 16.7) defines the custom `ListView` cell layout for this version of the Cover Viewer application. We broke the class into parts for discussion purposes. The `ImageTextCell` class extends `ListCell<Book>` because it defines a customized presentation of a `Book` in a `ListView` cell. For the best `ListView` performance, it's considered best practice to define the custom presentation's controls as instance variables in the `ListCell` subclass and configure them in the subclass's constructor. This minimizes the amount of work required in each call to method `updateItem`.

Click here to view code image

```
1  // Fig. 16.7: ImageTextCell.java
2  // Custom ListView cell factory that displays an Image and text
3  import javafx.geometry.Pos;
4  import javafx.scene.control.Label;
5  import javafx.scene.control.ListCell;
6  import javafx.scene.image.Image;
7  import javafx.scene.image.ImageView;
8  import javafx.scene.layout.VBox;
9  import javafx.scene.text.TextAlignment;
10
```

```
11   public class ImageTextCell extends ListCell<Book> {
12      private VBox vbox = new VBox(8.0); // 8 pixels of gap between controls
13      private ImageView thumbImageView = new ImageView(); // initially empty
14      private Label label = new Label();
15
```

Fig. 16.7 | Custom `ListView` cell factory that displays an image and text.

Constructor

The constructor (lines 17–29) configures the instance variables we use to build the custom presentation. Line 18 indicates that the `VBox`'s children should be centered. Lines 20–22 configure the `ImageView` and attach it to the `VBox`'s collection of children. Line 20 indicates that the `ImageView` should preserve the image's aspect ratio, and line 21 indicates that the `ImageView` should be 100 pixels tall. Line 22 attaches the `ImageView` to the `VBox`.

Click here to view code image

```
16      // constructor configures VBox, ImageView and Label
17      public ImageTextCell() {
18         vbox.setAlignment(Pos.CENTER); // center VBox contents horizontally
19
20         thumbImageView.setPreserveRatio(true);
21         thumbImageView.setFitHeight(100.0); // thumbnail 100 pixels tall
22         vbox.getChildren().add(thumbImageView); // attach to Vbox
23
24         label.setWrapText(true); // wrap if text too wide to fit in label
25         label.setTextAlignment(TextAlignment.CENTER); // center text
26         vbox.getChildren().add(label); // attach to VBox
27
28         setPrefWidth(USE_PREF_SIZE); // use preferred size for cell width
29      }
30
```

Lines 24–26 configure the `Label` and attach it to the `VBox`'s collection of children. Line 24 indicates that the `Label` should wrap its text if it's too wide to fit in the `Label`'s width, and line 25 indicates that the text should be centered in the `Label`. Line 26 attaches the `Label` to the `VBox`. Finally, line 28 indicates that the cell should use its preferred width, which is determined from the width of its parent `ListView`.

Method `updateItem`

Method `updateItem` (lines 32–46) configures the `Label`'s text and the `ImageView`'s `Image`, then displays the custom presentation in the `ListView`. The `ListView`'s cell factory calls this method when a `ListView` cell is required—when the `ListView` is first displayed and when `ListView` cells are about to scroll onto the screen. The method receives the `Book` to display and a `boolean` indicating whether the cell about to be created is empty. You must call the superclass's `updateItem` method (line 35) to ensure the custom cells display correctly.

Click here to view code image

```
31      // called to configure each custom ListView cell
32      @Override
33      protected void updateItem(Book item, boolean empty) {
34         // required to ensure that cell displays properly
35         super.updateItem(item, empty)
36
37         if (empty || item == null) {
38            setGraphic(null); // don't display anything
39         }
40         else {
41            // set ImageView's thumbnail image
42            thumbImageView.setImage(new Image(item.thumbImage()));
43            label.setText(item.title()); // configure Label's text
44            setGraphic(vbox); // attach custom layout to ListView cell
45         }
46      }
47   }
```

If the cell is empty or the item parameter is `null`, then there is no `Book` to display and line 38 calls the `ImageTextCell`'s inherited **`setGraphic` method** with `null`. This method receives the `Node` that should be displayed in the cell. Any JavaFX `Node` can be provided, giving you tremendous flexibility for customizing a cell's appearance.

If there is a `Book` to display, lines 40–45 configure the `ImageTextCell`'s `Label` and `ImageView`. Line 42 creates the `Book`'s `Image` object and sets it to display in the `ImageView`. Line 43 sets the `Label`'s text to the `Book`'s title. Finally, line 44 uses method `setGraphic` to set the `ImageTextCell`'s `VBox` as the custom cell's presentation.

16.6.4 `CoverViewerController` Class

Once you've defined the custom cell layout, updating the `CoverViewerController` to use it requires setting the `ListView`'s cell factory. Insert the following code as the last statement in the `CoverViewerController`'s `initialize` method:

Click here to view code image

```
// set custom ListView cell factory
booksListView.setCellFactory(
   listView -> new ImageTextCell();
);
```

The argument to `ListView` method `setCellFactory` is an implementation of the **functional interface** `CallBack` (package `javafx.util`). This generic interface provides a `call` method that receives one argument and returns a value. In this case, we implement interface `Callback` with a lambda. Though we don't use it here, the compiler will infer that the lambda's parameter is a `ListView<Book>`. The parameter represents the `ListView` in which the custom cells will appear. The compiler also infers that the lambda's return type is a `ListCell<Book>` object—an `ImageTextCell` *is a* `ListCell<Book>`. The lambda returns a new `ImageTextCell` object. Each time the `ListView` requires a new cell, the lambda is invoked to get a new `ImageTextCell`. Then the `ImageTextCell`'s `updateItem` method is called to create the custom cell presentation.

Generative AI

1 Prompt genAIs, asking them to explain JavaFX best practices and performance considerations for customizing a control's look-and-feel.

2 Prompt genAIs, asking for the advantages and disadvantages of creating JavaFX layouts programmatically compared to using Scene Builder and FXML.

16.7 `FileChooser` and `DirectoryChooser` Dialogs

JavaFX's **`FileChooser`** and **`DirectoryChooser`** **classes** (package `javafx.stage`) are predefined dialogs that applications can display so users can select local system files and directories. To demonstrate these, we'll enhance the file-and-directory processing example from Section 11.3. Sample screen captures are shown in the code discussion.

Creating the JavaFX GUI

The GUI (`FileChooserTest.fxml`) consists of a 600-by-400 `BorderPane` with the fx:id `borderPane`:

The `BorderPane`'s top contains a **`ToolBar`** **layout** (from the Scene Builder Library's Containers section), which arranges its controls horizontally (by default) or vertically. Typically, you place `ToolBar`s at your GUI's edges, such as in a `BorderPane`'s top, right, bottom or left areas.

The `BorderPane`'s center contains a **`TextArea`** **control** with the fx:id `textArea`. We set the control's Text property to `"Select file or directory"` and enabled its Wrap Text property to ensure that long lines of text wrap to the next line. If there are more lines of text to display than vertical lines in the `TextArea`, the control will show a vertical scrollbar. When Wrap Text is not enabled, the `TextArea` also shows a horizontal scrollbar if the text is too wide to display.

By default, the `ToolBar` you drag onto your layout has one `Button`. You can drag other controls onto the `ToolBar` and, if necessary, remove the default `Button`. We added a second `Button`. For the first `Button`, we set:

- the Text property to `"Select File"`,
- the fx:id property to `selectFileButton` and
- the On Action event handler to `selectFileButtonPressed`.

For the second `Button`, we set:

- the Text property to `"Select Directory"`,
- the fx:id property to `selectDirectoryButton` and
- the On Action event handler to `selectDirectoryButtonPressed`.

Finally, we specified `FileChooserTestController` as the FXML's controller.

Class That Launches the Application

Figure 16.8 shows the `FileChooserTest` subclass of `Application`, which operates the same as the previous examples' `Application` subclasses. The figure also shows the application's initial GUI.

Click here to view code image

```
1  // Fig. 16.8: FileChooserTest.java
2  // Application to test classes FileChooser and DirectoryChooser.
3  import javafx.application.Application;
4  import javafx.fxml.FXMLLoader;
5  import javafx.scene.Parent;
6  import javafx.scene.Scene;
7  import javafx.stage.Stage;
8
9  public class FileChooserTest extends Application {
10    @Override
11    public void start(Stage stage) throws Exception {
12       Parent root =
13          FXMLLoader.load(getClass().getResource("FileChooserTest.fxml"));
14
15       var scene = new Scene(root);
16       stage.setTitle("File Chooser Test"); // displayed in title bar
17       stage.setScene(scene);
18       stage.show();
19    }
20
21    public static void main(String[] args) {
22       launch(args);
23    }
24 }
```

Fig. 16.8 | Demonstrating `JFileChooser`.

Controller Class

Class `FileChooserTestController` (Fig. 16.9) responds to the `Button`s' events. We broke the class into parts for discussion purposes. Lines 18–21 declare the variables for the controls with fx:ids in the FXML file.

Click here to view code image

```
1  // Fig. 16.9: FileChooserTestController.java
2  // Displays information about a selected file or folder.
3  import java.io.File;
4  import java.io.IOException;
```

```
5  import java.nio.file.DirectoryStream;
6  import java.nio.file.Files;
7  import java.nio.file.Path;
8  import java.nio.file.Paths;
9  import javafx.event.ActionEvent;
10 import javafx.fxml.FXML;
11 import javafx.scene.control.Button;
12 import javafx.scene.control.TextArea;
13 import javafx.scene.layout.BorderPane;
14 import javafx.stage.DirectoryChooser;
15 import javafx.stage.FileChooser;
16
17 public class FileChooserTestController {
18    @FXML private BorderPane borderPane;
19    @FXML private Button selectFileButton;
20    @FXML private Button selectDirectoryButton;
21    @FXML private TextArea textArea;
22
```

Fig. 16.9 | Displays information about a selected file or folder.

Method `selectFileButtonPressed`

When the user presses the Select File button, method `selectFileButtonPressed` (lines 24–44) creates, configures and displays a `FileChooser` (shown in the first screen capture below the code). Line 28 sets the text displayed in the `FileChooser`'s title bar. Line 31 specifies the initial directory that should be opened when the `FileChooser` is displayed. Method `setInitialDirectory` receives a `File` object representing the directory's location—`"."` represents the current folder from which the application was launched.

Click here to view code image

```
23 // handles selectFileButton's event
24 @FXML
25 private void selectFileButtonPressed(ActionEvent e) {
26    // configure dialog allowing selection of a file
27    var fileChooser = new FileChooser();
28    fileChooser.setTitle("Select File");
29
30    // display files in folder from which the application was launched
31    fileChooser.setInitialDirectory(new File("."));
32
33    // display the FileChooser
34    File file = fileChooser.showOpenDialog(
35       borderPane.getScene().getWindow());
```

```
36
37      // process selected Path or display a message
38      if (file != null) {
39         analyzePath(file.toPath());
40      }
41      else {
42         textArea.setText("Select file or directory");
43      }
44   }
45
```

Lines 34–35 display the `FileChooser` by calling its **`showOpenDialog` method** to display a dialog with an Open button for opening a file. This method receives as its argument a reference to the application's `Window`. A non-`null` argument makes the `FileChooser` a modal dialog that prevents the user from interacting with the rest of the application until the dialog is dismissed—when the user selects a file or clicks Cancel. To obtain the application's `Window`, we use the `borderPane`'s `getScene` method to get a reference to its parent `Scene`, then use the `Scene`'s `getWindow` method to get a reference to the `Window` containing the `Scene`.

Method `showOpenDialog` returns a `File` representing the selected file's location, or `null` if the user clicks the Cancel button. If the `File` is not `null`, line 39 calls `analyzePath` to display the selected file's information—**`File` method `toPath`** returns a `Path` object representing the location. Otherwise, line 42 displays a message in the `TextArea` telling the user to select a file or directory. The screen captures following method `selectFileButtonPressed` show the `FileChooser` dialog with the `FileChooserTest.java` file selected and the file's information displayed in the application after the user presses the dialog's Open button. `FileChooser` also has a **`showSaveDialog` method** that displays a dialog with a Save button for saving a file.

Method `selectDirectoryButtonPressed`

When the user presses the Select Directory button, method `selectDirectoryButton-Pressed` (lines 47–67) creates, configures and displays a `DirectoryChooser`.

Click here to view code image

```
46      // handles selectDirectoryButton's event
47      @FXML
48      private void selectDirectoryButtonPressed(ActionEvent e) {
49         // configure dialog allowing selection of a directory
```

```
50          var directoryChooser = new DirectoryChooser();
51          directoryChooser.setTitle("Select Directory");
52
53          // display folder from which the application was launched
54          directoryChooser.setInitialDirectory(new File("."));
55
56          // display the DirectoryChooser
57          File file = directoryChooser.showDialog(
58             borderPane.getScene().getWindow());
59
60          // process selected Path or display a message
61          if (file != null) {
62             analyzePath(file.toPath());
63          }
64          else {
65             textArea.setText("Select file or directory");
66          }
67       }
68
```

The method performs the same tasks as method `selectFileButtonPressed`. The key difference is line 57, which calls **`DirectoryChooser` method `showDialog`** to display the dialog—there are not separate open and save dialogs for selecting folders. Method `show-Dialog` returns a `File` object representing the location of the selected directory, or `null` if the user clicks Cancel. If the `File` is not `null`, line 62 calls `analyzePath` to display information about the selected directory. Otherwise, line 65 displays a message in the `TextArea` telling the user to select a file or directory. The screen captures following method `select-FileButtonPressed` show the `FileChooser` dialog with the `FileChooserTest` directory selected and the directory's information displayed in the application after the user presses the Open button.

Method `analyzePath`

Both event handlers call method `analyzePath` (lines 70–110) to determine whether a `Path` is a file or directory, display information about it and, if it's a directory, list its contents.

Click here to view code image

```
69       // display information about file or directory user specifies
70       public void analyzePath(Path path) {
71          try {
72             // if the file or directory exists, display its info
73             if (path != null && Files.exists(path)) {
74                // gather file (or directory) information
```

```
75              var builder = new StringBuilder();
76              builder.append(String.format("%s:%n", path.getFileName()));
77              builder.append(String.format("%s a directory%n",
78                 Files.isDirectory(path) ? "Is" : "Is not"));
79              builder.append(String.format("%s an absolute path%n",
80                 path.isAbsolute() ? "Is" : "Is not"));
81              builder.append(String.format("Last modified: %s%n",
82                 Files.getLastModifiedTime(path)));
83              builder.append(String.format("Size: %s%n", Files.size(path)));
84              builder.append(String.format("Path: %s%n", path));
85              builder.append(String.format("Absolute path: %s%n",
86                 path.toAbsolutePath()));
87
88              if (Files.isDirectory(path)) { // output directory listing
89                 builder.append(String.format("%nDirectory contents:%n"));
90
91                 // object for iterating through a directory's contents
92                 DirectoryStream<Path> directoryStream =
93                    Files.newDirectoryStream(path);
94
95                 for (Path p : directoryStream) {
96                    builder.append(String.format("%s%n", p));
97                 }
98              }
99
100             // display file or directory info
101             textArea.setText(builder.toString());
102          }
103          else { // Path does not exist
104             textArea.setText("Path does not exist");
105          }
106       }
107       catch (IOException ioException) {
108          textArea.setText(ioException.toString());
109       }
110    }
111 }
```

Generative AI

1 Prompt genAIs, asking for an overview of all the dialog types provided by JavaFX and typical use-cases for each.

16.8 Other JavaFX Capabilities and JavaFX Accessibility

This section overviews various additional JavaFX capabilities.

`TableView` and `TreeTableView` Controls

Section 16.5 demonstrated how to bind data to a `ListView` control. You often load such data from a database (Chapter 20, Accessing Databases with JDBC and SQLite). JavaFX's `TableView` control (package `javafx.scene.control`) displays tabular data in rows and columns, and supports user interactions with that data. A `TreeTableView` control adds capabilities for hierarchical data with rows that can have child rows, such as a category and the items within that category.

Accessibility

JavaFX has accessibility features to help people with visual impairments use their devices. For example, the screen readers in various operating systems can speak screen text or text that you provide to help users with visual impairments understand the purpose of a control. Visually impaired users must enable their operating systems' screen-reading capabilities. JavaFX controls also support:

- GUI navigation via the keyboard—for example, the user can press the *Tab* key to jump from one control to the next. If a screen reader is enabled, as the user moves the focus from control to control, the screen reader will speak appropriate information about each control (discussed below).
- A high-contrast mode to make controls more readable—as with screen readers, visually impaired users must enable this feature in their operating systems.

See your operating system's documentation for information on enabling its screen reader and high-contrast mode.

Every JavaFX `Node` subclass also has the following accessibility-related properties:

- `accessibleTextProperty`—A `String` that a screen reader speaks for a control. For example, a screen reader usually speaks the text displayed on a `Button`, but setting this property for a `Button` causes the screen reader to speak this property's text instead. You can also set this property to

provide accessibility text for controls that do not have text, such as `ImageView`s.

- `accessibleHelpProperty`—A more detailed control description `String` than that provided by the `accessibleTextProperty`. This property's text should help the user understand the purpose of the control in the context of your application.
- `accessibleRoleProperty`—A value from the `enum AccessibleRole` (package `javafx.scene`). A screen reader uses this property value to determine the attributes and actions supported for a given control.
- `accessibleRoleDescriptionProperty`—A `String` text description of a control that a screen reader typically speaks followed by the control's contents (such as the text on a `Button`) or the value of the `accessibleTextProperty`.

In addition, you can add `Label`s to a GUI that describe other controls. In such cases, you should set each `Label`'s `labelFor` property to the specific control the `Label` describes. For example, a `TextField` in which the user can enter a phone number might be preceded by a `Label` containing the text `"Phone Number"`. If the `Label`'s `labelFor` property references the `TextField`, then a screen reader will read the `Label`'s text as well when describing the `TextField` to the user.

Creating Custom JavaFX Controls

You can create custom controls by extending existing JavaFX control classes to customize them or by extending JavaFX's `Control` class directly.

Generative AI

1 Prompt genAIs, asking them to discuss JavaFX's default accessibility features and best practices for making JavaFX GUIs accessible.

16.9 JavaFX Updates

This section overviews various JavaFX changes and enhancements.

Modularization

In Chapter 21, Java Platform Module System, we discuss Java's module system in detail. JavaFX 9 and higher are modularized, enabling developers to bundle only the JavaFX modules their apps require. The key JavaFX modules are:

- `javafx.base`—Contains the packages required by all JavaFX apps. All the other JavaFX modules depend on this one.
- `javafx.controls`—Contains the packages for controls, layouts and charts, including the various controls shown in this chapter and Chapter 15.
- `javafx.fxml`—Contains the packages for working with FXML, including the FXML features we demonstrated in this chapter and Chapter 15.
- `javafx.graphics`—Contains the packages for working with graphics, animation, CSS (for styling nodes), text and more (Chapter 17, JavaFX Graphics and Multimedia).
- `javafx.media`—Contains the packages for incorporating audio and video (Chapter 17, JavaFX Graphics and Multimedia).
- `javafx.swing`—Contains the packages for integrating older Swing GUI components into JavaFX apps.
- `javafx.web`—Contains the package for integrating web content.

Skinning APIs

Chapter 17 demonstrates formatting JavaFX `Node`s using Cascading Style Sheets (CSS)— originally for styling web page elements. CSS specifies presentation (e.g., fonts, spacing, sizes, colors, positioning) separately from the GUI's structure and content (layout containers, shapes, text, GUI components, etc.). If a JavaFX GUI's presentation is determined entirely by a style sheet, you can swap in a new style sheet—sometimes called a skin—to change the GUI's appearance. Each JavaFX control also has a skin class that determines its default appearance (located in **package** `javafx.scene.control.skin`). You can extend the appropriate skin class to customize the look-and-feel for a given type of control. You then create an object of your custom skin class and set it via a control's `setSkin` method.

Subscription API

JavaFX 21 added the new `Subscription` API for adding and removing event listeners pro-grammatically. See https://deitel.com/javafp5 for a demonstration.

High-DPI Screen Support

JavaFX supports high DPI (dots-per-inch) screens on Windows, macOS and Linux, as well as capabilities to programmatically manipulate the scale at which JavaFX apps are rendered.

Updated GStreamer

JavaFX implements its audio and video multimedia capabilities using the open-source GStreamer framework (https://gstreamer.freedesktop.org). JavaFX 23 incorporates a more recent version of GStreamer with bug fixes and performance enhancements.

Updated WebKit

JavaFX's `WebView` control enables you to embed web content in your JavaFX apps. It's based on the open-source WebKit framework (https://www.webkit.org)—a web browser engine that supports loading and rendering web pages. JavaFX incorporates WebKit updates several times per year.

CSS Transitions

JavaFX 23 added support for CSS transitions, which allow smoother animations when CSS properties change. We'll introduce CSS transitions in Section 17.7.

Generative AI

1 Prompt genAIs, asking them to summarize JavaFX updates in JavaFX versions 17–23.

16.10 JavaFX Resources and Libraries

JavaFX for Mobile and Embedded Devices

A key Java benefit is writing apps that can run on any device with a Java Virtual Machine (JVM), including notebook computers, desktop computers, servers, mobile devices and embedded devices (such as those used in the Internet of Things). Gluon Mobile provides a mobile-optimized JavaFX implementation for iOS, Android and devices like the inexpensive Raspberry Pi (https://www.raspberrypi.org/), which can be used as a stand-alone computer or for embedded-device applications. For more information, see

Click here to view code image

```
http://gluonhq.com/products/mobile/
```

Scenic View for Debugging JavaFX Scenes and Nodes

Scenic View is a debugging tool for JavaFX scenes and nodes. You embed Scenic View directly into your apps or run it as a standalone application. You can inspect and modify your JavaFX scenes and nodes dynamically to see how changes affect their presentation on the screen—without having to edit your code, recompile it and re-run it for each change. For more information, visit

Click here to view code image

```
https://github.com/JonathanGiles/scenic-view
```

JFX Central

The website JFX Central

```
https://www.jfx-central.com/
```

serves as a central hub for JavaFX developers, providing:

- Updates and announcements related to JavaFX, such as new releases, community events, and changes.
- A comprehensive list of libraries and tools available for JavaFX development.
- Step-by-step guides and tutorials for learning JavaFX.
- JavaFX examples with code demonstrating JavaFX features and best practices.

- Links to official documentation and resources, including the JavaFX API documentation and user guides.
- Links to articles and blog posts from the JavaFX community.
- JavaFX community information, such as forums, discussion groups and events.
- Information about JavaFX development tools.
- Links to JavaFX books, courses and videos for learning more about JavaFX.

Awesome JavaFX

Awesome JavaFX on GitHub

Click here to view code image

```
https://github.com/mhrimaz/AwesomeJavaFX
```

provides a lengthy list of JavaFX resources that includes links to:

- articles,
- tutorials and blog posts,
- key blogs and websites,
- YouTube videos,
- books (for purchase),
- many libraries, tools, projects and frameworks,
- slide shows from JavaFX presentations, and
- various real-world examples of JavaFX in use.

WebFX

The open-source WebFX project

```
https://webfx.dev/
```

enables you to convert JavaFX applications to pure JavaScript so they can execute in a web browser. Visit their demo page for examples of browser-based JavaFX apps:

```
https://webfx.dev/#/demos
```

You can download WebFX and see additional demos at:

Click here to view code image

```
https://github.com/webfx-project/webfx
```

Third-Party JavaFX Libraries

One popular open-source JavaFX library is ControlsFX

Click here to view code image

```
https://controlsfx.github.io/
https://github.com/controlsfx/controlsfx
```

which provides common dialogs, additional controls, validation capabilities, `TextField` enhancements, a `SpreadSheetView`, `TableView` enhancements and more. You can find the API documentation at

Click here to view code image

```
https://controlsfx.github.io/javadocs/
```

Their `FXSampler` demo app

Click here to view code image

```
https://controlsfx.github.io/fxsampler/
```

shows the library's features in action.

Generative AI

1 Prompt genAIs, asking them to locate JavaFX demo sites, resources and libraries.

16.11 Wrap-Up

In this chapter, we continued our presentation of JavaFX. We discussed JavaFX layout panes in more detail and used `BorderPane`, `TitledPane` and `Pane` to arrange controls.

We discussed the many mouse events supported by JavaFX nodes and used the `onMouseDragged` event in a simple Painter application that displayed `Circle`s as the user dragged the mouse across a `Pane`. We used `ToggleGroup`s and `RadioButton`s to manage mutually exclusive options and set the user data `Object` for a control. When

you selected a `RadioButton`, the Painter application obtained the selected `RadioButton` from the corresponding `ToggleGroup`, then accessed the `RadioButton`'s user data `Object` for convenient access to the Painter application's selected color or pen size.

We discussed property binding and property listeners then used them to implement a Color Chooser application. You bound a `TextField`'s text to a `Slider`'s value to automatically update the `TextField` when the user moved the `Slider`'s thumb. You also used a property listener to allow the application's controller to update a `Rectangle`'s color when a `Slider`'s value changed.

In the Cover Viewer application, we showed how to bind a JavaFX `ObservableList` collection to a `ListView` control to populate it with the collection's elements. By default, each object in the collection was displayed as a `String` in the `ListView`. You configured a property listener to display an image in an `ImageView` when the user selected an item in the `ListView`. We then modified the Cover Viewer application to use a custom `ListView` cell factory that specified a custom layout for the `ListView` cells' contents. We displayed `FileChooser` and `DirectoryChooser` dialogs so the user could select a file or directory.

Finally, we introduced several other JavaFX capabilities and recent changes, and discussed several JavaFX online resources. In the next chapter, we'll demonstrate additional JavaFX features, including graphics, multimedia and customizing a GUI's look-and-feel with JavaFX's CSS (Cascading Style Sheets) capabilities.

Special Section: Generative AI Exercises

For each of the following questions, try multiple genAIs to determine which performs best. Always verify the results. Keep in mind that they sometimes make mistakes and even hallucinate, creating unreliable information.

16.1 ***(Generative AI: Painter Application Enhancements)*** Prompt genAIs with the Painter application's FXML and code files and ask them for suggested enhancements. Implement the updates yourself, then ask the genAIs to generate the code for their suggestions. Ask them to make any GUI changes in their generated code to the FXML file, rather than program-matically creating new controls and layouts. Compare their generated code to yours.

16.2 ***(Generative AI: Property Bindings vs. Property Listeners)*** Section 15.5's Tip Calculator application used a property listener to

update the `tipPercentageLabel` when the user moved the `tipPercentageSlider`'s thumb. Prompt genAIs, asking them to discuss the differences between property bindings and property listeners, explain their pros and cons, and suggest when to prefer each approach.

16.3 ***(Generative AI: Bidirectional Property Bindings)*** Prompt genAIs for a tutorial on bidirectional property bindings. Next, provide them with Section 16.4's Color Chooser application FXML file and Java source code. Then, ask them to update the application to use bidirectional property bindings between each `Slider` and its corresponding `TextField` so the user can move the `Slider`'s thumb to update the `TextField` or type in the `TextField` to move the `Slider`'s thumb. Compile and run their updated code to confirm that it works.

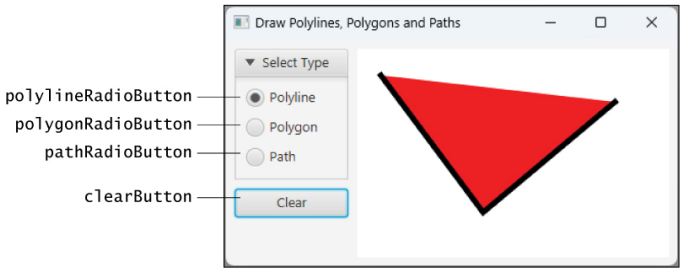
16.4 ***(Generative AI: JavaFX Application Lifecycle Events)*** Prompt genAIs to explain the JavaFX application lifecycle events and provide examples of handling them.

16.5 ***(Generative AI: JavaFX Internationalization and Localization)*** Prompt genAIs for a tutorial on localizing JavaFX applications so you can display them in the user's spoken language.

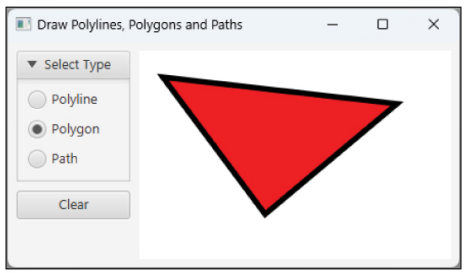
16.6 ***(Generative AI: JavaFX Responsive Design)*** Prompt genAIs for a tutorial on JavaFX responsive design. Then, provide them with the FXML and Java source code for any of your JavaFX applications, asking them how to update the GUI to adhere to responsive design principles.

16.7 ***(Generative AI: JavaFX Touch Events and Gestures)*** Prompt genAIs for a tutorial on JavaFX's support for touch events and gestures.

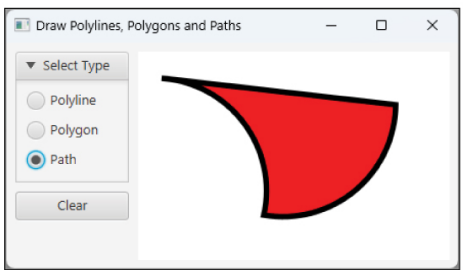
16.8 ***(Generative AI: JavaFX `InvalidationListener`s)*** Prompt genAIs for a tutorial on JavaFX `InvalidationListener`s and when to use them rather than `ChangeListener`s.



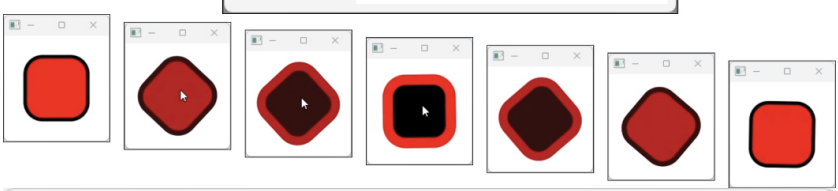
a) `Polyline` with three points—the starting and ending points are not connected



b) `Polygon` with three points—the starting and ending points are connected automatically



c) `Path` with two arc segments and a line connecting the first and last points



JavaFX Graphics and Multimedia

Objectives

In this chapter, you'll:

- Use JavaFX graphics and multimedia capabilities to make your apps "come alive" with graphics, animations, audio and video.
- Use external Cascading Style Sheets to customize the look of `Nodes` while maintaining their functionality.
- Customize font attributes like font family, size and style.
- Display two-dimensional `Arc`, `Circle`, `Ellipse`, `Line`, `Path`, `Polyline`, `Polygon` and `Rectangle` shape nodes.
- Customize a shape's stroke and fill using solid colors, images and gradients.
- Use `Transform`s to reposition and reorient nodes.
- Enable video playback with `Media`, `MediaPlayer` and `MediaView`.
- Animate `Node` properties with `Transition`s and `Timelines`.
- Use an `AnimationTimer` to create frame-based animations.
- Apply JavaFX CSS transitions to CSS properties.
- Draw graphics on a `Canvas`.
- Display 3D shapes.
- Be briefly introduced to FXGL for JavaFX game development.

Outline

17.1 Introduction

17.2 Controlling Fonts with Cascading Style Sheets (CSS)

17.2.1 CSS That Styles the GUI

17.2.2 FXML That Defines the GUI— Introduction to XML Markup

17.2.3 Referencing the CSS File from FXML
17.2.4 Specifying the `VBox`'s Style Class
17.2.5 Programmatically Loading CSS
17.3 Displaying Two-Dimensional Shapes
17.3.1 Defining Two-Dimensional Shapes with FXML
17.3.2 CSS That Styles the Two-Dimensional Shapes
17.4 `Polyline`s, `Polygon`s and `Path`s
17.4.1 GUI and CSS
17.4.2 `PolyShapesController` Class
17.5 Transforms
17.6 Playing Video with `Media`, `MediaPlayer` and `MediaView`
17.6.1 `VideoPlayer` GUI
17.6.2 `VideoPlayerController` Class
17.7 `Transition` Animations
17.7.1 `TransitionAnimations.fxml`
17.7.2 `TransitionAnimations-Controller` Class
17.8 `Timeline` Animations
17.9 Frame-by-Frame Animation with `AnimationTimer`
17.10 CSS Transitions
17.11 Drawing on a `Canvas`
17.12 Three-Dimensional Shapes
17.13 FXGL: A Brief Intro to Game Programming with JavaFX
17.14 Wrap-Up
Special Section: Generative AI Exercises

17.1 Introduction

In this chapter, we continue our discussion of JavaFX from Chapters 15 and 16. Here, we present various JavaFX graphics and multimedia capabilities. You'll:

- Use Cascading Style Sheets (CSS) to customize the appearance of JavaFX nodes.

- Customize fonts and font attributes used to display text.
- Display two-dimensional shapes, including lines, rectangles, circles, ellipses, arcs, polylines, polygons and custom paths.
- Apply transforms to `Node`s, such as rotating a `Node` around a particular point, scaling, translating (moving) and more.
- Display video and control its playback (e.g., play and pause).
- Dynamically modify JavaFX `Node`s with `Transition` and `Timeline` animations that change `Node` property values over time. As you'll learn, built-in `Transition` animations change specific JavaFX `Node` properties (such as a `Node`'s stroke and fill colors), but `Timeline` animations can change any modifiable `Node` property.
- Create frame-by-frame animations with an `AnimationTimer`.
- Apply CSS transitions to a `Node`'s CSS properties.
- Draw two-dimensional graphics on a `Canvas Node`.
- Display three-dimensional shapes, including boxes, cylinders and spheres.
- Get a brief introduction to FXGL for JavaFX game development.

This chapter does not include each example's `Application` subclass because it performs the same tasks we discussed and demonstrated in Chapters 15 and 16. Some examples do not have controller classes because they display JavaFX controls or graphics only to demonstrate Cascading Style Sheets (CSS) capabilities.

17.2 Controlling Fonts with Cascading Style Sheets (CSS)

In Chapters 15–16, you built JavaFX GUIs using Scene Builder. You specified a particular JavaFX object's appearance by selecting the object in Scene Builder then setting its property values in the Properties inspector. With this approach, if you want to change the GUI's appearance, you must edit each object. This can be time-consuming and error-prone for more complex GUIs in which you want to make the same changes to multiple objects.

In this chapter, we format JavaFX objects using **Cascading Style Sheets (CSS)**—a technology typically used to style web page

elements. A CSS style sheet specifies appearance details (fonts, spacing, sizes, colors, positioning, etc.) separately from the GUI's structure and content (layout containers, shapes, text, GUI components, etc.) If a JavaFX GUI's presentation is determined entirely by CSS rules, you update its appearance by swapping in a new style sheet.

In this section, you'll use CSS to specify several `Label`s' font properties and the `VBox` layout's spacing and padding properties. You'll place **CSS rules** that specify the font properties, spacing and padding in a separate file that ends with the **`.css` filename extension**, then reference that file from the FXML. You'll observe that

- before referencing the CSS file from the FXML, Scene Builder displays the GUI without styling and
- after referencing the CSS file from the FXML, Scene Builder renders the GUI with the CSS rules applied to the appropriate objects.

For a complete reference containing

- all the JavaFX CSS properties,
- the JavaFX `Node` types to which the attributes can be applied, and
- the allowed values for each attribute

visit:

Click here to view code image

```
https://openjfx.io/javadoc/23/javafx.graphics/javafx/scene/doc-
   files/cssref.html
```

17.2.1 CSS That Styles the GUI

Figure 17.1 presents this app's CSS rules specifying the `VBox`'s and `Label`s' styles. We split the file into parts for discussion purposes. We use the shorthand "app" for "application" throughout this chapter.

`.vbox` CSS Rule—Style Class Selectors

Lines 4–7 define the `.vbox` CSS rule that will be applied to this

app's `VBox` object (lines 8–18 of Fig. 17.2).

Click here to view code image

```
1  /* Fig. 17.1: FontsCSS.css */
2  /* CSS rules that style the VBox and Labels */
3
4  .vbox {
5     -fx-spacing: 10;
6     -fx-padding: 10;
7  }
8
```

Fig. 17.1 | CSS rules that style the `VBox` and `Label`s.

Each CSS rule begins with a **CSS selector** which specifies the JavaFX objects that will be styled according to the rule. In this rule, `.vbox` is a **style class selector**. The CSS properties in this rule are applied to any JavaFX object that has a `styleClass` property with the value `"vbox"`. In CSS, a style class selector begins with a dot (`.`) and is followed by its **class name** (not to be confused with a Java class). By convention, selector names typically have all lowercase letters, and multi-word names separate each word from the next with a dash (`-`).

Each CSS rule's body is delimited by required braces (`{}`) containing the CSS properties applied to objects matching the CSS selector. JavaFX CSS property names begin with `-fx-` followed by the JavaFX object's property name in all lowercase letters.[1] So,

1. According to `https://openjfx.io/javadoc/23/javafx.graphics/javafx/scene/doc-files/cssref.html`, JavaFX CSS properties are considered extensions to CSS, so JavaFX's CSS property names begin with "`-fx-`" to ensure they have distinct names from their HTML CSS counterparts.

- **`-fx-spacing`** (line 5) defines the value for a JavaFX object's `spacing` property and
- **`-fx-padding`** (line 6) defines the value for a JavaFX object's `padding` property.

Each property value is specified to the right of a required colon (`:`). In this case, we set

- `-fx-spacing` to `10` for 10 pixels of vertical space between objects in the `VBox` and
- `-fx-padding` to `10` to separate the `VBox`'s contents from the

`VBox`'s edges by 10 pixels at the top, right, bottom and left edges.

You also can specify the `-fx-padding` with four values separated by spaces. For example,

```
-fx-padding: 10 5 10 5
```

specifies 10 pixels for the top padding, 5 for the right, 10 for the bottom and 5 for the left. In Section 17.2.2, you'll apply the `.vbox` CSS rule to the `VBox` object.

`#label1` CSS Rule—ID Selectors

Lines 9–11 of Fig. 17.1 define the `#label1` CSS rule that will be applied to the `Label` with the `fx:id label1`.

Click here to view code image

```
 9  #label1 {
10     -fx-font: bold 14pt Arial;
11  }
12
```

Selectors that begin with `#` are **ID selectors**—they're applied to objects with the specified ID. In this case, the `#label1` selector matches an object with the `fx:id label1`—that is, the `Label` object in line 12 of Fig. 17.2. The `#label1` CSS rule

```
-fx-font: bold 14pt Arial;
```

sets an object's `font` property so that a corresponding object displays its text in a bold, 14-point Arial font. The `-fx-font` property can specify all aspects of a font, including its style, weight, size and font family. The size and font family are required. There are also properties for each aspect of a font: **`-fx-font-style`**, **`-fx-font-weight`**, **`-fx-font-size`** and **`-fx-font-family`**. These are applied to a JavaFX object's similarly named properties. For more information on CSS font attributes, visit

Click here to view code image

```
https://openjfx.io/javadoc/23/javafx.graphics/javafx/scene/doc-
   files/cssref.html#typefont
```

For a complete list of CSS selector types and how you can combine

them, visit

Click here to view code image

```
https://www.w3.org/TR/selectors-3/
```

`#label2` CSS Rule

Lines 13–15 of Fig. 17.1 define the `#label2` CSS rule that will be applied to the `Label` with the `fx:id label2`.

Click here to view code image

```
13  #label2 {
14     -fx-font: 16pt "Times New Roman";
15  }
16
```

The CSS property

Click here to view code image

```
-fx-font: 16pt "Times New Roman";
```

specifies only the required font size (`16pt`) and font family (`"Times New Roman"`) components—font family names with multiple words must be enclosed in quotes (`""`).

`#label3` CSS Rule

Lines 17–19 define the `#label3` CSS rule that will be applied to the `Label` with the `fx:id label3`.

Click here to view code image

```
17  #label3 {
18     -fx-font: bold italic 16pt "Courier New";
19  }
20
```

The CSS property

Click here to view code image

```
-fx-font: bold italic 16pt "Courier New";
```

specifies all the font components—weight (`bold`), style (`italic`), size (`16pt`) and font family (`"Courier New"`).

`#label4` CSS Rule

Lines 21–24 define the `#label4` CSS rule that will be applied to the `Label` with the `fx:id label4`.

```
21   #label4 {
22      -fx-font-size: 14pt;
23      -fx-underline: true;
24   }
25
```

The CSS property

```
-fx-font-size: 14pt;
```

specifies the font size `14pt`—all other aspects of this `Label`'s font are inherited from the `Label`'s parent container. The CSS property

```
-fx-underline: true;
```

indicates that the text in the `Label` should be underlined—this property's default value is `false`.

`#label5` CSS Rule

Lines 26–28 define the `#label5` CSS rule that will be applied to the `Label` with the `fx:id label5` and set its font size to `14pt`.

```
26   #label5 {
27      -fx-font-size: 14pt;
28   }
29
```

`#label5 .text` CSS Rule

Lines 30–32 define the `#label5 .text` CSS rule that will be applied to the `Text` object within the `Label` that has the `fx:id` value `"label5"`.

Click here to view code image

```
30   #label5 .text {
31      -fx-strikethrough: true;
32   }
```

This selector combines an ID selector and a style class selector.

Each `Label` contains a `Text` object with the CSS class `.text`. When applying this CSS rule, JavaFX first locates the object with the ID `label5`, then within that object, looks for a nested object that specifies the class `text`.

The CSS property

```
-fx-strikethrough: true;
```

indicates that the text in the `Label` should be displayed with a line through it—this property's default value is `false`.

17.2.2 FXML That Defines the GUI—Introduction to XML Markup

Figure 17.2 presents `FontCSS.fxml`. The `FontCSS` app's FXML GUI consists of a `VBox` layout element (lines 8–18) containing five `Label`s (lines 12–16). To create this in Scene Builder, drag five `Label`s onto the `VBox`. Configure their text as in Fig. 17.2(a) and set their fx:ids from top to bottom as `label1`, `label2`, `label3`, `label4` and `label5`. All the `Label`s initially have the same appearance, and there's no spacing between or around them.[2]

2. In many of this chapter's examples, after creating a GUI in Scene Builder, we used a text editor to format the FXML, remove unnecessary properties that were inserted by Scene Builder and properties that we specified via CSS rules. The text editors in various popular IDEs also can be used to edit FXML. For this reason, when you build these examples from scratch, your FXML may differ from what's presented in this chapter. You also can set a property to its default value in Scene Builder to remove it from the FXML.

Click here to view code image

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!-- Fig. 17.2: FontCSS.fxml -->
3  <!-- FontCSS GUI that is styled via external CSS -->
4
5  <?import javafx.scene.control.Label?>
6  <?import javafx.scene.layout.VBox?>
7
8  <VBox styleClass="vbox" stylesheets="@FontCSS.css"
9     xmlns="http://javafx.com/javafx/23"
10    xmlns:fx="http://javafx.com/fxml/1">
11    <children>
12       <Label fx:id="label1" text="Arial 14pt bold" />
13       <Label fx:id="label2" text="Times New Roman 16pt plain" />
```

```
14          <Label fx:id="label3" text="Courier New 16pt bold and italic" />
15          <Label fx:id="label4" text="Default font 14pt with underline" />
16          <Label fx:id="label5" text="Default font 14pt with strikethrough" />
17       </children>
18    </VBox>
```

Fig. 17.2 | `FontCSS` GUI that is styled via external CSS.

XML Declaration

Each FXML document begins with an **XML declaration** (line 1), which must be the first line in the file and indicates that the document contains XML markup. For FXML documents, line 1 must appear as in Fig. 17.2. The XML declaration's **`version` attribute** specifies the document's XML syntax version (`1.0`). The **`encoding` attribute** specifies the document's character format—XML documents typically contain Unicode characters in `UTF-8` format.[3] Scene Builder automatically inserts the XML declaration.

3. "UTF-8." Wikipedia, Wikimedia Foundation. Accessed September 26, 2024. https://en.wikipedia.org/wiki/UTF-8.

Attributes

Each XML attribute has the format

name="*value*"

The *name* and *value* are separated by `=`, and the *value* is placed in quotes (`""`). Multiple *name*=*value* pairs are separated by whitespace.

Comments

Lines 2–3 are XML comments. These are delimited by `<!--` and `-->` and can be placed almost anywhere in an XML document. XML comments can span multiple lines. We added these XML comments to state the FXML document's figure number, filename and purpose, as we typically do in our Java code.

FXML `import` Declarations

Lines 5–6 are **FXML `import` declarations**—delimited by `<?import` and `?>`—specifying the fully qualified names of the JavaFX types used in the document. Scene Builder automatically inserts FXML `import` declarations based on the various JavaFX node types you

add to the GUI.

Elements

XML documents contain **elements** specifying the document's structure. Most elements are delimited by a **start tag** and an **end tag**:

- A start tag consists of **angle brackets** (`<` and `>`) containing the element's name followed by zero or more attributes. For example, the `VBox` element's start tag (lines 8–10) has four attributes.
- An end tag consists of the element name preceded by a **forward slash** (`/`) in angle brackets—for example, `</VBox>` in line 18.

An element's start and end tags enclose the element's contents. In this case, lines 11–17 declare other elements that describe the `VBox`'s contents. Every XML document must have exactly one **root element** containing the other elements. In Fig. 17.2, `VBox` is the root.

A layout element always contains a `children` element (lines 11–17) that, in turn, contains the child `Node`s arranged by that layout. For a `VBox`, the `children` element's child `Node`s are listed in their display order from top to bottom. The elements in lines 12–16 represent the `VBox`'s five `Label`s. These are **empty elements** that use the shorthand start-tag-only notation:

```
<ElementName attributes />
```

in which the empty element's start tag ends with `/>` rather than `>`. The empty element:

Click here to view code image

```
<Label fx:id="label1" text="Arial 14pt bold"/>
```

is equivalent to

Click here to view code image

```
<Label fx:id="label1" text="Arial 14pt bold">
</Label>
```

which does not have content between the start and end tags. Empty elements often have attributes (such as `fx:id` and `text` for each

`Label` element).

XML Namespaces

In lines 9–10, the `VBox` attributes

Click here to view code image

```
xmlns="http://javafx.com/javafx/23"
xmlns:fx="http://javafx.com/fxml/1"
```

specify the XML namespaces used in FXML markup. An XML **namespace** specifies a collection of element and attribute names you can use in the document. The attribute

Click here to view code image

```
xmlns="http://javafx.com/javafx/23"
```

specifies the default namespace. FXML `import` declarations (like those in lines 5–6) add names to this namespace for use in the document. The attribute

Click here to view code image

```
xmlns:fx="http://javafx.com/fxml/1"
```

specifies JavaFX's `fx` namespace. Elements and attributes from this namespace (such as the `fx:id` attribute) are used internally by the `FXMLLoader` class. For example, for each FXML element that specifies an `fx:id`, the `FXMLLoader` initializes a corresponding variable in the controller class. The `fx:` in `fx:id` is a **namespace prefix** that specifies the namespace (`fx`) that defines the attribute (`id`). Every element or attribute name in Fig. 17.2 that does not begin with `fx:` is part of the default namespace.

17.2.3 Referencing the CSS File from FXML

For the `Label`s to appear with the customized fonts, we must reference the `FontCSS.css` file from the FXML. This enables Scene Builder to apply the CSS rules to the GUI. To reference the CSS file:

- Select the `VBox` in the Scene Builder.
- In the Properties inspector, click the + button under the Stylesheets heading.
- In the dialog that appears, select the `FontCSS.css` file and

click Open.

This adds the `stylesheets` attribute (line 8)

```
stylesheets="@FontCSS.css"
```

to the `VBox`'s opening tag (lines 8–10). The `@` symbol—called the local resolution operator in FXML—indicates that `FontCSS.css` is located relative to the FXML file on disk. The CSS and FXML files must be in the same folder if no path information is specified.

17.2.4 Specifying the `VBox`'s Style Class

The preceding steps apply the font styles to the `Label`s based on their ID selectors but do not apply the spacing and padding to the `VBox`. For the `VBox`, we defined a CSS rule using a style class selector named `.vbox`. To apply this CSS rule to the `VBox`:

- Select the `VBox` in the Scene Builder.
- In the Properties inspector, under the Style Class heading, click the empty box and select `vbox` from the list of available selectors.

This adds the `styleClass` attribute

```
styleClass="vbox"
```

to the `VBox`'s opening tag (line 8). At this point, the GUI appears as in Fig. 17.2(b). You can now run the app to produce the output in Fig. 17.2(c).

17.2.5 Programmatically Loading CSS

In the `FontCSS` app, the FXML referenced the CSS style sheet directly (line 8). It's also possible to load CSS files dynamically and add them to a `Scene`'s collection of style sheets. You might do this, for example, in an app that enables users to choose their preferred look-and-feel, such as a light background with dark text vs. a dark background with light text.

To load a stylesheet dynamically, add the following statement to the `Application` subclass's `start` method:

Click here to view code image

```
scene.getStylesheets().add(
   FontCSS.class.getResource("FontCSS.css").toExternalForm());
```

In the preceding statement:

- The notation `FontCSS.class` creates a `Class` object representing the `FontCSS` subclass of `Application`. This is equivalent to calling the method `getClass()`, which all classes inherit.
- `Class` method `getResource` returns a `URL` representing the location of the file `FontCSS.css`. Method `getResource` looks for the file in the same location from which class `FontCSS` was loaded by the JVM.
- `URL` method `toExternalForm` returns the `URL`'s `String` representation. This is passed to the `add` method of the `Scene`'s collection of style sheets to add the style sheet to the scene.

Method `getStyleSheets` returns an `ObservableList` object, which supports standard `List` methods, so you may also `remove` a stylesheet and `clear` all the stylesheets programmatically.

Generative AI

1 Prompt genAIs, asking them to write a tutorial on the JavaFX CSS selector types and the ways in which they can be combined.

2 Prompt genAIs, asking them to write a tutorial on JavaFX hierarchical CSS selectors and pseudoclasses with examples of how to use them.

17.3 Displaying Two-Dimensional Shapes

JavaFX has two ways to draw shapes:

- You can define **`Shape`** and **`Shape3D`** (package `javafx.scene.shape`) subclass objects, add them to a container in the JavaFX stage and manipulate them like other JavaFX `Node`s.
- You can add a **`Canvas`** object (package **`javafx.scene.canvas`**) to a container in the JavaFX stage, then draw on it using various `GraphicsContext` methods.

This section's `BasicShapes` example displays **`Line`**, **`Rectangle`**, **`Circle`**, **`Ellipse`** and **`Arc`** `Shape`s. Like other `Node` types, you can drag these from the Scene Builder Library's Shapes category onto the design area, then configure them via the Inspector's Properties, Layout and Code sections. You can also create objects of any JavaFX `Node` type programmatically.

17.3.1 Defining Two-Dimensional Shapes with FXML

Figure 17.3 contains the completed FXML for the `BasicShapes` app, which references the `BasicShapes.css` file (line 13) that we present in Section 17.3.2. For this app, we used Scene Builder to drag two `Line`s, a `Rectangle`, a `Circle`, an `Ellipse` and an `Arc` onto a `Pane` layout, then configured their dimensions and positions.

Click here to view code image

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!-- Fig. 17.3: BasicShapes.fxml -->
3  <!-- Defining Shape objects and styling via CSS -->
4
5  <?import javafx.scene.layout.Pane?>
6  <?import javafx.scene.shape.Arc?>
7  <?import javafx.scene.shape.Circle?>
8  <?import javafx.scene.shape.Ellipse?>
9  <?import javafx.scene.shape.Line?>
10 <?import javafx.scene.shape.Rectangle?>
11
12 <Pane id="Pane" prefHeight="110.0" prefWidth="630.0"
13    stylesheets="@BasicShapes.css" xmlns="http://javafx.com/javafx/23"
14    xmlns:fx="http://javafx.com/fxml/1">
15    <children>
16       <Line fx:id="line1" endX="100.0" endY="100.0"
17          startX="10.0" startY="10.0" />
18       <Line fx:id="line2" endX="10.0" endY="100.0"
19          startX="100.0" startY="10.0" />
20       <Rectangle fx:id="rectangle" height="90.0" layoutX="120.0"
21          layoutY="10.0" width="90.0" />
22       <Circle fx:id="circle" centerX="270.0" centerY="55.0"
23          radius="45.0" />
24       <Ellipse fx:id="ellipse" centerX="430.0" centerY="55.0"
25          radiusX="100.0" radiusY="45.0" />
26       <Arc fx:id="arc" centerX="590.0" centerY="55.0" length="270.0"
27          radiusX="45.0" radiusY="45.0" startAngle="45.0" type="ROUND" />
28    </children>
29 </Pane>
```

Fig. 17.3 | Defining `Shape` objects and styling via CSS.

Each property you can set in Scene Builder has a corresponding attribute in FXML. For example, the `Pane` object's Pref Height property in Scene Builder corresponds to the `prefHeight` attribute (line 12) in FXML. When you build this GUI in Scene Builder, use the FXML attribute values in Fig. 17.3. As you drag each shape onto your design, Scene Builder automatically configures certain properties, such as the Fill and Stroke colors for the `Rectangle`, `Circle`, `Ellipse` and `Arc`. For each property that does not have a corresponding attribute in Fig. 17.3, you can remove the attribute by setting it to its default value in Scene Builder or by manually editing the FXML.

Lines 6–10 import the shape classes used in the FXML. We also specified `fx:id` values (lines 16 and 18) for the two `Line`s—we use these values in CSS rules with ID selectors to define separate styles for each `Line`. We removed the shapes' `fill`, `stroke` and `strokeType` properties that Scene Builder autogenerated. The default `fill` for a shape is black. The default stroke is a one-pixel black line. The default `strokeType` is centered—based on the stroke's thickness, half the thickness appears inside the shape's bounds and half outside. You also may display a shape's stroke entirely inside or outside the shape's bounds. We specify the strokes and fills with the styles in Section 17.3.2.

`Line` Objects

Lines 16–17 and 18–19 define two `Line`s. Each connects two endpoints specified by the properties **`startX`**, **`startY`**, **`endX`** and **`endY`**. The x- and y-coordinates are measured from the top-left corner of the `Pane`, with x-coordinates increasing from left to right and y-coordinates increasing from top to bottom. If you specify a `Line`'s **`layoutX`** and **`layoutY`** properties, the `startX`, `startY`, `endX` and `endY` properties are measured from that point.

`Rectangle` Object

Lines 20–21 define a `Rectangle` object. A `Rectangle` is displayed based on its `layoutX`, `layoutY`, **`width`** and **`height`** properties:

- A `Rectangle`'s upper-left corner is positioned at the coordinates specified by the `layoutX` and `layoutY` properties, which are inherited from class `Node`.
- A `Rectangle`'s dimensions are specified by the `width` and

`height` properties—in this case, they have the same value, so the `Rectangle` defines a square.

`Circle` Object

Lines 22–23 define a `Circle` object with its center at the point specified by the **`centerX`** and **`centerY`** properties. The **`radius`** property determines the `Circle`'s size (two times the `radius`) around its center point.

`Ellipse` Object

Lines 24–25 define an `Ellipse` object. Like a `Circle`, an `Ellipse`'s center is specified by the **`centerX`** and **`centerY`** properties. You also specify **`radiusX`** and **`radiusY`** properties that help determine the `Ellipse`'s width (left and right of the center point) and height (above and below the center point).

`Arc` Object

Lines 26–27 define an `Arc` object. Like an `Ellipse`, an `Arc`'s center is specified by the **`centerX`** and **`centerY`** properties, and the **`radiusX`** and **`radiusY`** properties determine the `Arc`'s width and height. For an `Arc`, you also specify:

- **`length`**—The arc's length in degrees (0–360). Positive values sweep counterclockwise.
- **`startAngle`**—The angle in degrees at which the arc should begin.
- **`type`**—How the arc should be closed. `ROUND` indicates that the starting and ending points of the arc should be connected to the center point by straight lines. You also may choose `OPEN`, which does not connect the start and end points, or `CHORD`, which connects the start and end points with a straight line.

17.3.2 CSS That Styles the Two-Dimensional Shapes

Figure 17.4 contains the CSS for the `BasicShapes` app. In this CSS file, we define two CSS rules with ID selectors (`#line1` and `#line2`) to style the app's two `Line` objects. The remaining rules use **type selectors**, which apply to all objects of a given type. You specify a type selector by using the JavaFX class name. We split the file into

parts for discussion purposes.

Specifying Common Attributes for Various Objects

The CSS rule in lines 4–6 defines the `-fx-stroke-width` CSS property for all the shapes in the app—this property specifies the thickness of the `Line`s and the border thickness of all the other shapes:

Click here to view code image

```
1  /* Fig. 17.4: BasicShapes.css */
2  /* CSS that styles various two-dimensional shapes */
3
4  Line, Rectangle, Circle, Ellipse, Arc {
5      -fx-stroke-width: 10;
6  }
7
```

Fig. 17.4 | CSS that styles various two-dimensional shapes.

We use CSS type selectors in a comma-separated list to apply this rule to multiple shapes. Line 4 indicates this should be applied to all objects of types `Line`, `Rectangle`, `Circle`, `Ellipse` and `Arc`.

Styling the First `Line`

The CSS rule in lines 8–10 sets the `-fx-stroke` to the solid color `red` for the `Line` with the `fx:id "line1"`:

```
8  #line1 {
9      -fx-stroke: red;
10 }
11
```

This rule is in addition to the one at lines 4–6, which sets the stroke width for all the shapes in this app. When JavaFX renders an object, it combines all the CSS rules that apply to the object to determine its appearance. Colors may be specified as

- named colors (such as `"red"`, `"green"` and `"blue"`),
- colors defined by their red, green, blue and alpha (transparency) components,
- colors defined by their hue, saturation, brightness and alpha components,

and more. For details on all the ways to specify color in CSS, visit

Click here to view code image

```
https://openjfx.io/javadoc/23/javafx.graphics/javafx/scene/doc-
   files/cssref.html#typecolor
```

Styling the Second `Line`

The CSS rule in lines 12–15 applies to the `Line` with the `fx:id "line2"`.

Click here to view code image

```
12  #line2 {
13      -fx-stroke: rgba(0%, 50%, 0%, 0.5);
14      -fx-stroke-line-cap: round;
15  }
16
```

For this rule, we specified the `-fx-stroke` property's color using the CSS function **`rgba`**, which defines a color based on its red, green, blue and alpha (transparency) components. We used the version of `rgba` that receives percentages from 0% to 100%, specifying the amount of red, green and blue in the color and a value from 0.0 (transparent) to 1.0 (opaque) for the alpha component. Line 13 produces a semitransparent green line. Notice the interaction between the two `Line`s' colors at the intersection point in Fig. 17.3's output windows. The **`-fx-stroke-line-cap`** CSS property (line 14) indicates that the ends of the `Line` should be rounded—the rounding effect is more noticeable with thicker strokes.

Styling the `Rectangle`

For `Rectangle`s, `Circle`s, `Ellipse`s and `Arc`s, you can specify both the `-fx-stroke` for the shapes' borders and the **`-fx-fill`** for the color or pattern that appears inside the shape. The rule in lines 17–22 uses a CSS type selector to indicate that all `Rectangle`s should have `red` borders (line 18) and `yellow` fill (line 21).

```
17  Rectangle {
18     -fx-stroke: red;
19     -fx-arc-width: 50;
20     -fx-arc-height: 50;
21     -fx-fill: yellow;
22  }
```

Lines 19–20 define the `Rectangle`'s **`-fx-arc-width`** and **`-fx-arc-height`** properties. These specify the width and height of an ellipse that's divided in half horizontally and vertically, then used to round the `Rectangle`'s corners. Because these properties have the same value (`50`) in this app, the four corners are each a quarter of a circle with a diameter of 50.

Styling the `Circle`

The CSS rule in lines 24–27 applies to all `Circle` objects.

Click here to view code image

```
24  Circle {
25     -fx-stroke: blue;
26     -fx-fill: radial-gradient(center 50% 50%, radius 60%, white, red);
27  }
28
```

Line 25 sets the `Circle`'s stroke to `blue`. Line 26 sets the `Circle`'s fill with a **gradient**—colors that transition gradually from one color to the next. You can transition between as many colors as you like and specify the points at which to change colors—such points are called **color stops**. You can use gradients for any property that specifies a color. In this case, we use the CSS function **`radial-gradient`** to create a gradient in which the color changes gradually from a center point outward. The fill

Click here to view code image

```
-fx-fill: radial-gradient(center 50% 50%, radius 60%, white, red);
```

indicates the gradient should begin from a `center` point at `50% 50%`—the shape's horizontal and vertical middle. The `radius` specifies the distance from the center at which an even mixture of the two colors appears. This radial gradient begins with `white` in the center and ends with `red` at the outer edge of the `Circle`'s fill. We'll discuss a linear gradient momentarily.

Styling the `Ellipse`

The CSS rule in lines 29–32 applies to all `Ellipse` objects.

Click here to view code image

```
29  Ellipse {
30     -fx-stroke: green;
31     -fx-fill: image-pattern("yellowflowers.png");
32  }
33
```

Line 30 specifies that an `Ellipse` should have a `green` stroke. Line 31 specifies that the `Ellipse`'s fill should be the image in `yellowflowers.png` (located in this app's folder). If you're building this app from scratch, copy the image from the `images` folder with the chapter's examples into the app's folder. To specify an image as fill, you use the CSS function **`image-pattern`**.[4]

4. At the time of this writing, Scene Builder does not display a shape's fill correctly if it's specified with a CSS image-pattern. You must run the example to display the fill (Fig. 17.3(b)). We'll post updates on the book's web site: `https://deitel.com/javafp5`.

Styling the `Arc`

The CSS rule in lines 34–37 applies to all `Arc` objects.

Click here to view code image

```
34  Arc {
35     -fx-stroke: purple;
36     -fx-fill: linear-gradient(to right, cyan, white);
37  }
```

Line 35 specifies that an `Arc` should have a `purple` stroke. In this case, line 36

Click here to view code image

```
-fx-fill: linear-gradient(to right, cyan, white);
```

specifies that the `Arc` should be filled with a **linear gradient**—such gradients gradually transition from one color to the next horizontally, vertically or diagonally. You can transition between as many colors as you like and specify the points at which to change colors. To create a linear gradient, you use the CSS function **`linear-gradient`**. In this case,

```
to right
```

indicates that the gradient should start from the shape's left edge and transition through colors to the shape's right edge. We specified

only two colors here—`cyan` at the left edge of the gradient and `white` at the right edge—but two or more colors can be specified in the comma-separated list. For more information on all the options for configuring radial gradients, linear gradients and image patterns, visit

Click here to view code image

```
https://openjfx.io/javadoc/23/javafx.graphics/javafx/scene/doc-
   files/cssref.html#typepaint
```

Generative AI

1 Prompt genAIs for an overview of the JavaFX CSS options for configuring radial gradients, linear gradients and image patterns.

2 Prompt genAIs for an overview of JavaFX CSS options for the `-fx-stroke-line-cap` property.

3 Prompt genAIs for a list of color names you can use in JavaFX CSS.

4 Prompt genAIs for an overview of creating custom colors in JavaFX CSS.

17.4 `Polyline`s, `Polygon`s and `Path`s

Several JavaFX classes enable you to create custom shapes:

- **`Polyline`**—draws a series of connected lines defined by a collection of points.
- **`Polygon`**—draws a series of connected lines defined by a set of points and connects the last point to the first point.
- **`Path`**—draws a series of connected **`PathElement`s** by moving to a given point, then drawing lines, arcs and curves.

In the `PolyShapes` app, you select the shape to display by clicking a `RadioButton` in the left column. You specify a shape's points by clicking throughout the `AnchoredPane` in which the shapes are displayed.

17.4.1 GUI and CSS

The following screen captures contain the `PolyShapes` app's GUI. It's similar to the `Painter` app in Section 16.3, so we show only the key GUI elements' `fx:id` property values rather than the complete FXML. Each `fx:id` property value ends with the GUI element's type. In this GUI:

- The three `RadioButton`s are in a `ToggleGroup` with the `fx:id` `"toggleGroup"`. The Polyline `RadioButton` is Selected by default. We also set each `RadioButton`'s On Action event handler to `shapeRadioButtonSelected`.
- We dragged a `Polyline`, a `Polygon` and a `Path` from the Scene Builder Library's Shapes section onto the `Pane` that displays the shapes, and we set their `fx:id`s to `polyline`, `polygon` and `path`, respectively. We set each shape's `visible` property to `false` by selecting the shape in Scene Builder then unchecking the Visible checkbox in the Properties inspector. We display only the shape with the selected `RadioButton` at runtime.
- We set the `Pane`'s On Mouse Clicked event handler to `drawingAreaMouseClicked`.
- We set the Clear `Button`'s On Action event handler to `clearButtonPressed`.
- We set the controller class to `PolyShapesController`.
- Finally, we edited the FXML to remove the `Path` object's `<elements>` and the `Polyline` and `Polygon` objects' `<points>`, as we'll set these programmatically in response to the user's mouse-click events.

The `PolyShapes.css` file defines the properties `-fx-stroke`, `-fx-stroke-width` and `-fx-fill` that are applied to all three shapes in this example. Each shape has a black, five-pixel outline and with red fill color:

```
Polyline, Polygon, Path {
   -fx-stroke: black;
   -fx-stroke-width: 5;
   -fx-fill: red;
}
```

17.4.2 `PolyShapesController` Class

Class `PolyShapesController` (Fig. 17.5) responds to this app's user

interactions. We split the code into parts for discussion purposes. The `enum ShapeType` (line 17) defines constants we use to determine which shape to display.

Click here to view code image

```
1  // Fig. 17.5: PolyShapesController.java
2  // Drawing Polylines, Polygons and Paths.
3  import javafx.event.ActionEvent;
4  import javafx.fxml.FXML;
5  import javafx.scene.control.RadioButton;
6  import javafx.scene.control.ToggleGroup;
7  import javafx.scene.input.MouseEvent;
8  import javafx.scene.shape.ArcTo;
9  import javafx.scene.shape.ClosePath;
10 import javafx.scene.shape.MoveTo;
11 import javafx.scene.shape.Path;
12 import javafx.scene.shape.Polygon;
13 import javafx.scene.shape.Polyline;
14
15 public class PolyShapesController {
16    // enum representing shape types
17    private enum ShapeType {POLYLINE, POLYGON, PATH};
18
```

Fig. 17.5 | Drawing `Polyline`s, `Polygon`s and `Path`s.

Lines 20–26 declare the variables corresponding to the GUI components and shapes with `fx:id`s in the FXML. The `shapeType` variable (line 29) stores whichever shape type is currently selected in the GUI's `RadioButton`s. By default, the `Polyline` will be displayed. The `sweepFlag` variable will determine whether to draw an arc with a negative or positive sweep angle.

Click here to view code image

```
19    // instance variables that refer to GUI components
20    @FXML private RadioButton polylineRadioButton;
21    @FXML private RadioButton polygonRadioButton;
22    @FXML private RadioButton pathRadioButton;
23    @FXML private ToggleGroup toggleGroup;
24    @FXML private Polyline polyline;
25    @FXML private Polygon polygon;
26    @FXML private Path path;
27
28    // instance variables for managing state
29    private ShapeType shapeType = ShapeType.POLYLINE;
30    private boolean sweepFlag = true; // used with arcs in a Path
31
```

Method `initialize`

Recall from Section 16.3.1 that you can associate any `Object` with each JavaFX control via its `setUserData` method. For the shape `RadioButton`s in this app, method `initialize` (lines 33–41) stores the `ShapeType` that each `RadioButton` represents (lines 36–38). When handling the `RadioButton` events, we use these to set the `shapeType` instance variable. Line 40 calls method `displayShape` to display the currently selected shape (the `Polyline` by default). Initially, the shape is not visible because it has no points.

Click here to view code image

```
32      // set user data for the RadioButtons and display polyline object
33      @FXML
34      private void initialize() {
35         // user data on a control can be any Object
36         polylineRadioButton.setUserData(ShapeType.POLYLINE);
37         polygonRadioButton.setUserData(ShapeType.POLYGON);
38         pathRadioButton.setUserData(ShapeType.PATH);
39
40         displayShape(); // sets polyline's visibility to true when app loads
41      }
42
```

Method `drawingAreaMouseClicked`

When the user clicks the app's `Pane`, method `drawingAreaMouseClicked` (lines 44–65) modifies all three shapes to incorporate the new point at which the user clicked.

Click here to view code image

```
43      // handles drawingArea's onMouseClicked event
44      @FXML
45      private void drawingAreaMouseClicked(MouseEvent e) {
46         polyline.getPoints().addAll(e.getX(), e.getY());
47         polygon.getPoints().addAll(e.getX(), e.getY());
48
49         // if path is empty, move to first click position and close path
50         if (path.getElements().isEmpty()) {
51            path.getElements().add(new MoveTo(e.getX(), e.getY()));
52            path.getElements().add(new ClosePath());
53         }
54         else { // insert a new path segment before the ClosePath element
55            // create an arc segment and insert it in the path
56            ArcTo arcTo = new ArcTo();
57            arcTo.setX(e.getX());
```

```
58              arcTo.setY(e.getY());
59              arcTo.setRadiusX(100.0);
60              arcTo.setRadiusY(100.0);
61              arcTo.setSweepFlag(sweepFlag);
62              sweepFlag = !sweepFlag;
63              path.getElements().add(path.getElements().size() - 1, arcTo);
64          }
65      }
66
```

`Polyline`s and `Polygon`s store their points as a collection of `Double`s—the first two values represent the first point's location, the next two represent the second point's location, etc. Line 46 adds the new click point to the `polyline` object's points collection by calling `addAll` with the `MouseEvent`'s x- and y-coordinate values. This adds the new point's information to the end of the collection. Line 47 performs the same task for the `polygon` object.

Lines 50–64 manipulate the `path` object. A `Path` is represented by a collection of `PathElement`s. The subclasses of `PathElement` used in this example are:

- **`MoveTo`**—Moves to a specific position without drawing anything.
- **`ArcTo`**—Draws an arc from the previous `PathElement`'s endpoint to the specified location. We'll discuss this in more detail momentarily.
- **`ClosePath`**—Closes the path by drawing a straight line from the last `PathElement`'s endpoint to the first `PathElement` start point.

Other `PathElement`s not covered here include `LineTo`, `HLineTo`, `VLineTo`, `CubicCurveTo` and `QuadCurveTo`.

When the user clicks the `Pane`, line 50 checks whether the `Path` contains elements. If not, line 51 moves the `path`'s starting point to the mouse-click location by adding a `MoveTo` element to the `path`'s `PathElement`s collection. Then, line 52 adds a new `Close-Path` element to complete the path. For each subsequent mouse-click event, lines 56–61 create an `ArcTo` element, and line 63 inserts it before the `ClosePath` element by calling the `PathElement`s collection's `add` method that receives an index as its first argument.

Lines 57–58 set the `ArcTo` element's endpoint to the `MouseEvent`'s coordinates. The arc is drawn as a piece of an ellipse for which you specify the horizontal radius and vertical radius

(lines 59–60). Line 61 sets the `ArcTo`'s `sweepFlag`, which determines whether the arc sweeps in the positive angle direction (`true`; counter-clockwise) or the negative angle direction (`false`; clockwise). By default, an `ArcTo` element is drawn as the shortest arc between the last `PathElement`'s endpoint and the point specified by the `ArcTo` element. To sweep the arc the long way around the ellipse, set the `ArcTo`'s `largeArcFlag` to `true`. For each mouse click, line 62 reverses the value of our controller class's `sweepFlag` instance variable so that the `ArcTo` elements toggle between positive and negative angles for variety.

Method `shapeRadioButtonSelected`

When the user clicks a shape `RadioButton`, lines 71–72 set the controller's `shapeType` instance variable, then line 73 calls method `displayShape` to display the selected shape. Try creating a `Polyline` of several points, then changing to the `Polygon` and `Path` to observe how the points are used in each shape.

Click here to view code image

```
67      // handles color RadioButton's ActionEvents
68      @FXML
69      private void shapeRadioButtonSelected(ActionEvent e) {
70         // user data for each color RadioButton is a ShapeType constant
71         shapeType =
72            (ShapeType) toggleGroup.getSelectedToggle().getUserData();
73         displayShape(); // display the currently selected shape
74      }
75
```

Method `displayShape`

Lines 78–80 in method `displayShape` set the visibility of the three shapes based on the current `shapeType`. The currently selected shape's visibility is set to `true` to display the shape, and the other shapes' visibility is set to `false` to hide those shapes.

Click here to view code image

```
76      // displays currently selected shape
77      private void displayShape() {
78         polyline.setVisible(shapeType == ShapeType.POLYLINE);
79         polygon.setVisible(shapeType == ShapeType.POLYGON);
80         path.setVisible(shapeType == ShapeType.PATH);
81      }
```

Method `clearButtonPressed`

When the user clicks the Clear `Button`, lines 86–87 in method `clearButtonPressed` clear the `polyline`'s and `polygon`'s collections of points, and line 88 clears the `path`'s collection of `PathElement`s. The user can then begin drawing a new shape by clicking the `Pane`.

Click here to view code image

```
83      // resets each shape
84      @FXML
85      private void clearButtonPressed(ActionEvent event) {
86         polyline.getPoints().clear();
87         polygon.getPoints().clear();
88         path.getElements().clear();
89      }
90   }
```

Generative AI

1 Prompt genAIs for the difference between closing and not closing a JavaFX `Path`.

17.5 Transforms

A **transform** can be applied to any UI element to reposition or reorient the element. The built-in JavaFX transforms are subclasses of **`Transform`**. Some of these subclasses include:

- **`Translate`**—moves an object to a new location.
- **`Rotate`**—rotates an object around a point and by a specified rotation angle.
- **`Scale`**—scales an object's size by the specified amounts.

The next example draws stars using the `Polygon` control and uses `Rotate` transforms to create a circle of randomly colored stars. This app's FXML consists of an empty 300-by-300 `Pane` layout with the `fx:id "pane"`. We also set the controller class to `DrawStarsController`. Figure 17.6 contains the app's controller and a sample output.

Click here to view code image

```
1  // Fig. 17.6: DrawStarsController.java
2  // Create a circle of stars using Polygons and Rotate transforms
3  import java.util.random.RandomGenerator;
4  import javafx.fxml.FXML;
5  import javafx.scene.layout.Pane;
6  import javafx.scene.paint.Color;
7  import javafx.scene.shape.Polygon;
8  import javafx.scene.transform.Transform;
9
10 public class DrawStarsController {
11    private static final RandomGenerator random =
12       RandomGenerator.getDefault();
13    @FXML private Pane pane;
14
15    @FXML
16    private void initialize() {
17       // points that define a five-pointed star shape
18       Double[] points = {205.0,150.0, 217.0,186.0, 259.0,186.0,
19          223.0,204.0, 233.0,246.0, 205.0,222.0, 177.0,246.0, 187.0,204.0,
20          151.0,186.0, 193.0,186.0};
21
22       // create 18 stars
23       for (int count = 0; count < 18; ++count) {
24          // create a new Polygon and copy existing points into it
25          var newStar = new Polygon();
26          newStar.getPoints().addAll(points);
27
28          // create random Color and set as newStar's fill
29          newStar.setStroke(Color.GREY);
30          newStar.setFill(Color.rgb(random.nextInt(256),
31             random.nextInt(256), random.nextInt(256),
32             random.nextDouble()));
33
34          // apply a rotation to the shape
35          newStar.getTransforms().add(
36             Transform.rotate(count * 20, 150, 150));
37          pane.getChildren().add(newStar);
38       }
39    }
40 }
```

Fig. 17.6 | Create a circle of stars using `Polygon`s and `Rotate` transforms.

Method `initialize` (lines 15–39) defines the stars, applies the transforms and attaches the stars to the app's `pane`. Lines 18–20 define the ten *x*-*y* coordinates that form a star as a `Double` array. A

`Polygon`'s collection of points is a generic collection, so you must use type `Double` rather than `double`—recall that primitive types cannot be used in Java generic collections. Each pair of values in the array represents a point's *x*-*y* coordinates.

During each iteration of the loop, lines 25–36 create a `Polygon` using the points in the `points` array and apply a `Rotate` transform, resulting in a circle of `Polygon`s. To generate each star's random colors, we use a `RandomGenerator` to create three random values from 0–255 for the red, green and blue components of the color and one random value from 0.0–1.0 for the color's alpha transparency value. We pass those values to class `Color`'s `static rgb` method to create a `Color`.

To apply a rotation to the new `Polygon`, we add a `Rotate` transform to its collection of `Transform`s (lines 35–36). **`Transform`'s `static` method `rotate`** (line 36) returns a `Rotate` object. The method's first argument is the rotation angle. Each iteration of the loop assigns a new rotation-angle value by using the control variable multiplied by 20 as the `rotate` method's first argument. The method's next two arguments are the *x*-*y* coordinates of the point of rotation around which the `Polygon` rotates. The center of the circle of stars is the point *(150, 150)*—we rotated all 18 stars around that point. Adding each `Polygon` as a new child element of the `pane` object (line 37) allows the `Polygon` to be rendered on screen.

Generative AI

1 Prompt genAIs, asking them to list all the JavaFX `Transform`s with explanations of what they do.

2 Prompt genAIs, asking how JavaFX `Transform`s can be combined.

17.6 Playing Video with `Media`, `MediaPlayer` and `MediaView`

Many of today's most popular apps are multimedia intensive. JavaFX provides audio and video multimedia capabilities via the **`javafx.scene.media`** package:

- For simple audio playback, use class **`AudioClip`**.
- For audio playback with more playback controls and for video playback, use classes **`Media`**, **`MediaPlayer`** and **`MediaView`**.

In this section, you'll build a basic video player.[5] We'll explain `Media`, `MediaPlayer` and `MediaView` as we encounter them in this example's controller class (Section 17.6.2). The video we use in this example is from NASA's multimedia library[6] and was downloaded from

5. At the time of this writing, the MP4 video in this example would not load correctly into the JavaFX app running on Windows 11 ARM and would load occasionally on Windows 11 x64. We'll post updates to the book's web page: `https://deitel.com/javafp5`.
6. For NASA's terms of use, visit `https://www.nasa.gov/nasa-brand-center/images-and-media/`

Click here to view code image

```
https://images.nasa.gov/details/A1Launch
```

We included the video file `A1Launch~medium.mp4` in the `video` folder with this chapter's examples. When building the app from scratch, copy the video into the app's folder.

Media Formats

JavaFX supports several video formats, but some are platform-specific. MPEG-4 video (also called MP4) is well-supported across platforms. If you find a video in a format that's not supported, there are many free online video conversion tools you can use to convert a video file to another format, including:

Click here to view code image

```
https://convertio.co/video-converter/
```

and

```
https://cloudconvert.com/
```

ControlsFX Library's `ExceptionDialog`

`ExceptionDialog` is one of many additional JavaFX controls available through the open-source project ControlsFX at

Click here to view code image

```
https://controlsfx.github.io/
https://github.com/controlsfx/controlsfx
```

We use an `ExceptionDialog` in this app to display a message to the user if an error occurs during media playback.

For your convenience, we include the latest version of the ControlsFX JAR file at the time of this writing

```
controlsfx-11.2.1.jar
```

in this chapter's examples folder. For convenience, place this file in your JavaFX SDK folder's `lib` subfolder.

Compiling and Running the App with ControlsFX

To compile and run this app, you must include ControlsFX and the JavaFX media capabilities in the `javac` and `java` commands. Enter the entire following command, then press or select *Enter* to compile the code in the `VideoPlayer` folder:

Click here to view code image

```
javac --module-path pathToFX --add-modules
   javafx.controls,javafx.fxml,javafx.media,org.controlsfx.controls
   *.java
```

Similarly, to run the app, enter the entire following command, then press or select *Enter*:

Click here to view code image

```
java --module-path pathToFX --add-modules
   javafx.controls,javafx.fxml,javafx.media,org.controlsfx.controls
   VideoPlayer
```

Recall from Section 15.5 that you need to replace *pathToFX* with

- `$PATH_TO_FX` if you are using macOS,
- `$env:PATH_TO_FX` if you are using a Microsoft Windows PowerShell, or
- `%PATH_TO_FX%` if you are using a Microsoft Windows Command Prompt.

17.6.1 VideoPlayer GUI

Figure 17.7 contains the completed `VideoPlayer.fxml` file and two sample screen captures of the running `VideoPlayer` app. The GUI's layout is a `BorderPane` consisting of

- a `MediaView` (from the Scene Builder Library's Controls section) with the `fx:id mediaView` and
- a `ToolBar` (from the Scene Builder Library's Containers section) containing one `Button` with the `fx:id playPauseButton` and the text `"Play"`. The controller method `playPauseButtonPressed` responds when the `Button` is pressed.

We placed the `MediaView` in the `BorderPane`'s center region (lines 25–27) so that it occupies all available space in the `BorderPane`, and we placed the `ToolBar` in the `BorderPane`'s `bottom` region (lines 15–24). By default, Scene Builder adds one `Button` to the `ToolBar` when you drag the `ToolBar` onto your layout. You can then add other controls to the `ToolBar` as necessary. We set the controller class to `VideoPlayerController`.

Click here to view code image

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!-- Fig. 17.7: VideoPlayer.fxml -->
3  <!-- VideoPlayer GUI with a MediaView and a Button. -->
4
5  <?import javafx.scene.control.Button?>
6  <?import javafx.scene.control.ToolBar?>
7  <?import javafx.scene.layout.BorderPane?>
8  <?import javafx.scene.media.MediaView?>
9
10 <BorderPane prefHeight="400.0" prefWidth="600.0"
11    style="-fx-background-color: black;"
12    xmlns="http://javafx.com/javafx/23"
13    xmlns:fx="http://javafx.com/fxml/1"
14    fx:controller="VideoPlayerController">
15    <bottom>
16       <ToolBar prefHeight="40.0" prefWidth="200.0"
17          BorderPane.alignment="CENTER">
18          <items>
19             <Button fx:id="playPauseButton"
20                onAction="#playPauseButtonPressed" prefHeight="25.0"
21                prefWidth="60.0" text="Play" />
22          </items>
23       </ToolBar>
```

```
24      </bottom>
25      <center>
26         <MediaView fx:id="mediaView" BorderPane.alignment="CENTER" />
27      </center>
28   </BorderPane>
```

Fig. 17.7 | `VideoPlayer` GUI with a `MediaView` and a `Button`. Video courtesy of NASA—visit `https://www.nasa.gov/nasa-brand-center/images-and-media/` for usage rules.

17.6.2 `VideoPlayerController` Class

Class `VideoPlayerController` (Fig. 17.8) configures video playback, responds to the `MediaPlayer`'s state changes and responds to the `playPauseButton`'s events. The controller uses classes `Media`, `MediaPlayer` and `MediaView` as follows:

- A `Media` object specifies the location of the media to play and provides access to various information about the media, such as its duration, video dimensions and more.
- A `MediaPlayer` object loads a `Media` object and controls its playback, transitioning through various states (ready, playing, paused, etc.) during loading and playback. You'll use lambdas to define `Runnable`s that respond to these state transitions.
- A `MediaView` object is a `Node` in the scene graph that displays the `Media` being played by a given `MediaPlayer` object.

We split the code into parts for discussion purposes.

Click here to view code image

```
1   // Fig. 17.8: VideoPlayerController.java
2   // Using Media, MediaPlayer and MediaView to play a video.
3   import java.net.URL;
4   import javafx.beans.binding.Bindings;
5   import javafx.beans.property.DoubleProperty;
6   import javafx.event.ActionEvent;
7   import javafx.fxml.FXML;
8   import javafx.scene.control.Button;
9   import javafx.scene.media.Media;
10  import javafx.scene.media.MediaPlayer;
11  import javafx.scene.media.MediaPlayer.Status;
12  import javafx.scene.media.MediaView;
13  import javafx.util.Duration;
```

```
14  import org.controlsfx.dialog.ExceptionDialog;
15
16  public class VideoPlayerController {
```

Fig. 17.8 | Using `Media`, `MediaPlayer` and `MediaView` to play a video.

Instance Variables

Lines 17–19 declare the controller's instance variables. When the app loads, `mediaView` (line 17) is assigned a reference to the `MediaView` object declared in the app's FXML. The `mediaPlayer` variable (line 19) is configured in method `initialize` to load the video specified by a `Media` object and used by method `playPauseButtonPressed` (lines 67–77) to play and pause the video.

Click here to view code image

```
17  @FXML private MediaView mediaView;
18  @FXML private Button playPauseButton;
19  private MediaPlayer mediaPlayer;
20
```

Creating a `Media` Object Representing the Video to Play

Method `initialize` (lines 21–64) configures media playback, registers event handlers for `MediaPlayer` events and sets up a data binding to change the Button's text to Play or Pause based on the `MediaPlayer`'s state. Lines 24–25 get a `URL` representing the location of the video file. The notation

```
VideoPlayerController.class
```

is a `Class` object representing the `VideoPlayerController` class. This is equivalent to calling the inherited method `getClass()`. Next, line 28 creates a `Media` object representing the video. The argument to the `Media` constructor is a `String` representing the video's location, which we obtain with `URL` method `toExternalForm`. The URL `String` can represent a local file on your computer or can be a location on the web. The `Media` constructor throws various exceptions, including `MediaException`s if the media cannot be found or is not of a supported media format.

Click here to view code image

```
21      @FXML
22      private void initialize() {
23         // get URL of the video file
24         URL url = VideoPlayerController.class.getResource(
25            "A1Launch~medium.mp4");
26
27         // create a Media object for the specified URL
28         var media = new Media(url.toExternalForm());
29
30         // create a MediaPlayer to control Media playback
31         mediaPlayer = new MediaPlayer(media);
32
33         // specify which MediaPlayer to display in the MediaView
34         mediaView.setMediaPlayer(mediaPlayer);
35
36         // set handler to be called when the video completes playing
37         mediaPlayer.setOnEndOfMedia(() -> {
38            mediaPlayer.seek(Duration.ZERO);
39            mediaPlayer.pause();
40         });
41
42         // set handler that displays an ExceptionDialog if an error occurs
43         mediaPlayer.setOnError(() -> {
44            var dialog = new ExceptionDialog(mediaPlayer.getError());
45            dialog.showAndWait();
46         });
47
48         // set handler to resize mediaView to window size once ready to play
49         mediaPlayer.setOnReady(() -> {
50            DoubleProperty width = mediaView.fitWidthProperty();
51            DoubleProperty height = mediaView.fitHeightProperty();
52            width.bind(Bindings.selectDouble(
53               mediaView.sceneProperty(), "width"));
54            height.bind(Bindings.selectDouble(
55               mediaView.sceneProperty(), "height"));
56         });
57
58         // bind playPauseButton's text to the MediaPlayer's status
59         playPauseButton.textProperty().bind(Bindings.when(
60            mediaPlayer.statusProperty().isEqualTo(Status.PLAYING))
61                        .then("Pause")
62                        .otherwise("Play")
63         );
64      }
65
```

Creating a `MediaPlayer` Object to Load the Video and Control Playback

To load the video and prepare it for playback, you must associate it with a `MediaPlayer` object (line 31). Playing multiple videos requires a separate `MediaPlayer` for each `Media` object. However, a given `Media` object can be associated with multiple `MediaPlayer`s. The `MediaPlayer` constructor throws

- a `NullPointerException` if the `Media` is `null` or
- a `MediaException` if a problem occurs while constructing the `MediaPlayer`.

Attaching the `MediaPlayer` Object to the `MediaView` to Display the Video

To display video, you must associate a `MediaPlayer` with a `MediaView`. When the `MediaView` already exists—e.g., when it's created in FXML—you call the **`MediaView`'s `setMediaPlayer` method** (line 34) to perform this task. When creating a `MediaView` object programmatically, you can pass the `MediaPlayer` to the `MediaView`'s constructor. A `MediaView` is like any other `Node` in the scene graph, so you can apply CSS styles, transforms and animations (Sections 17.7–17.9) to it as well.

Configuring Event Handlers for `MediaPlayer` Events

A `MediaPlayer` transitions through various states. Common states include ready, playing and paused. For these and other states, you can execute a task as the `MediaPlayer` enters the corresponding state. In addition, you can specify tasks that execute when the end of media playback is reached or when an error occurs during playback. To perform a task for a given state, you specify an object that implements the **`Runnable` interface** (package `java.lang`). This interface contains a no-parameter `run` method that returns `void`.

`Runnable` is a functional interface that can be implemented using a lambda. For example, lines 37–40 call the **`MediaPlayer`'s `setOnEndOfMedia` method**, passing a lambda that executes when video playback completes. Line 38 calls **`MediaPlayer` method `seek`** to move to the beginning of the video, and line 39 pauses the video.

Lines 43–46 call the **`MediaPlayer`'s `setOnError` method** to specify a task to perform if the `MediaPlayer` enters the error state, indicating that an error occurred during playback. In this case, we display a ControlsFX `ExceptionDialog` containing the error message. Calling the `ExceptionDialog`'s `showAndWait` method indicates that the app must wait for the user to dismiss the dialog

before continuing.

Binding the `MediaView`'s Size to the Scene's Size

Lines 49–56 call the **`MediaPlayer`'s `setOnReady` method** to specify a task to perform if the `MediaPlayer` enters the ready state. We use property bindings to bind the `MediaView`'s `width` and `height` to the scene's `width` and `height` so it resizes when the app's window resizes. A `Node`'s `sceneProperty` returns an object of type **`ReadOnlyObjectProperty<Scene>`** you can use to access the `Scene` in which the `Node` is displayed. This object has many properties. To bind to its specific properties, you can use the methods of **class `Bindings`** (package `javafx.beans.binding`) to select the corresponding properties. The `Scene`'s `width` and `height` are each `DoubleProperty` objects. **`Bindings` method `selectDouble`** gets a reference to a `DoubleProperty`. The method's first argument is the object that contains the property, and the second argument is the name of the property to which you'd like to bind.

Binding the `Button`'s Text to the `MediaPlayer`'s Status

Lines 59–63 bind the `Button`'s text to the `MediaPlayer`'s `Status`, so the text updates based on the current playback status. `Button` method `textProperty` returns the button's text property, which we `bind` to the result of a condition configured using the **`Bindings` method `when`**. This method's argument (line 60) gets the `MediaPlayer`'s status property and calls its `isEqualTo` method to compare the current status to `Status.PLAYING`. The `when` call returns a `When` object that specifies what to do based on the condition. If the condition is `true`, the `Button`'s text is set to `"Pause"` via the chained call `.then("Pause")`; otherwise, the `Button`'s text is set to `"Play"` via the chained call `.otherwise("Play")`.

Method `playPauseButtonPressed`

The event handler `playPauseButtonPressed` (lines 67–77) toggles video playback. We get the `MediaPlayer`'s `Status` (line 69), then when the status is either `Status.READY` (ready to play) or `Status.PAUSED` (currently paused), line 72 calls the **`MediaPlayer`'s `play` method**; otherwise, line 75 sets calls the **`MediaPlayer`'s `pause` method**.

Click here to view code image

```
66      // toggle media playback and the text on the playPauseButton
67      @FXML
68      private void playPauseButtonPressed(ActionEvent e) {
69         Status status = mediaPlayer.getStatus();
70
71         if (status == Status.READY || status == Status.PAUSED) {
72            mediaPlayer.play();
73         }
74         else {
75            mediaPlayer.pause();
76         }
77      }
78   }
```

Generative AI

1 Prompt genAIs, asking for common data bindings implemented with the JavaFX `Bindings` method `when`.

2 Prompt genAIs for a brief tutorial with examples of other data bindings you can configure using the JavaFX `Bindings` class.

17.7 `Transition` Animations

Animations in JavaFX apps transition a `Node`'s property values from one value to another in a specified amount of time. Most properties of a `Node` can be animated. This section focuses on several of JavaFX's predefined `Transition` animations from the **`javafx.animations` package**. By default, the subclasses that define `Transition` animations change the values of specific `Node` properties. For example, a `FadeTransition` changes the value of a `Node`'s `opacity` property (which specifies whether the `Node` is opaque or transparent) over time, whereas a `PathTransition` changes a `Node`'s location by moving it along a `Path` over time. Though we include sample screen captures for all the animation examples, the best way to experience each is to run the examples yourself.

17.7.1 `TransitionAnimations.fxml`

Figure 17.9 contains this app's GUI and sample screen captures of it running. When you click the `Button` (lines 17–18), its `startButtonPressed` event handler in the app's controller creates a

sequence of `Transition` animations for the `Rectangle` (lines 15–16) and plays them. The `Rectangle` is styled with the following CSS from the file `TransitionAnimations.css`:

```
Rectangle {
   -fx-stroke-width: 10;
   -fx-stroke: red;
   -fx-arc-width: 50;
   -fx-arc-height: 50;
   -fx-fill: yellow;
}
```

which produces a rounded rectangle with a 10-pixel red border and yellow fill.

Click here to view code image

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!-- Fig. 17.9: TransitionAnimations.fxml -->
3  <!-- FXML for a Rectangle and Button -->
4
5  <?import javafx.scene.control.Button?>
6  <?import javafx.scene.layout.Pane?>
7  <?import javafx.scene.shape.Rectangle?>
8
9  <Pane id="Pane" prefHeight="200.0" prefWidth="180.0"
10    stylesheets="@TransitionAnimations.css"
11    xmlns="http://javafx.com/javafx/23"
12    xmlns:fx="http://javafx.com/fxml/1"
13    fx:controller="TransitionAnimationsController">
14    <children>
15       <Rectangle fx:id="rectangle" height="90.0" layoutX="45.0"
16          layoutY="45.0" width="90.0" />
17       <Button layoutX="38.0" layoutY="161.0" mnemonicParsing="false"
18          onAction="#startButtonPressed" text="Start Animations" />
19    </children>
20 </Pane>
```

Fig. 17.9 | FXML for a `Rectangle` and `Button`.

17.7.2 `TransitionAnimationsController` Class

Class `TransitionAnimationsController` (Fig. 17.10) defines the `startButton`'s event handler (lines 25–81). This event handler defines several animations that are played in sequence. We split the code—including the method `startButtonPressed`—into parts for discussion purposes.

Click here to view code image

```
1  // Fig. 17.10: TransitionAnimationsController.java
2  // Applying Transition animations to a Rectangle.
3  import javafx.animation.FadeTransition;
4  import javafx.animation.FillTransition;
5  import javafx.animation.Interpolator;
6  import javafx.animation.ParallelTransition;
7  import javafx.animation.PathTransition;
8  import javafx.animation.RotateTransition;
9  import javafx.animation.ScaleTransition;
10 import javafx.animation.SequentialTransition;
11 import javafx.animation.StrokeTransition;
12 import javafx.event.ActionEvent;
13 import javafx.fxml.FXML;
14 import javafx.scene.paint.Color;
15 import javafx.scene.shape.LineTo;
16 import javafx.scene.shape.MoveTo;
17 import javafx.scene.shape.Path;
18 import javafx.scene.shape.Rectangle;
19 import javafx.util.Duration;
20
21 public class TransitionAnimationsController {
22    @FXML private Rectangle rectangle;
23
```

Fig. 17.10 | Applying `Transition` animations to a `Rectangle`.

`FillTransition`

Lines 28–33 configure a one-second **`FillTransition`** that changes a shape's fill color.

Click here to view code image

```
24    // configure and start transition animations
25    @FXML
26    private void startButtonPressed(ActionEvent event) {
27       // transition that changes a shape's fill
28       var fillTransition = new FillTransition(Duration.seconds(1));
29       fillTransition.setToValue(Color.CYAN);
30       fillTransition.setCycleCount(2);
31
32       // each even cycle plays transition in reverse to restore original
33       fillTransition.setAutoReverse(true);
34
```

Line 29 specifies the color (`CYAN`) to which the fill will transition.

Line 30 sets the animation's cycle count to 2—this determines the transition's number of iterations to perform over the specified duration. Line 33 specifies that the animation should automatically play itself in reverse once the initial transition is complete. For this animation, during the first cycle, the fill color changes from the original fill color to `CYAN`, and during the second cycle, the animation transitions back to the original fill color.

`StrokeTransition`

Lines 36–39 configure a one-second **`StrokeTransition`** that changes a shape's stroke color.

Click here to view code image

```
35          // transition that changes a shape's stroke over time
36          var strokeTransition = new StrokeTransition(Duration.seconds(1));
37          strokeTransition.setToValue(Color.BLUE);
38          strokeTransition.setCycleCount(2);
39          strokeTransition.setAutoReverse(true);
40
```

Line 37 specifies the color (`BLUE`) to which the stroke will transition. Line 38 sets the animation's cycle count to 2, and line 39 specifies that the animation should automatically play itself in reverse once the initial transition is complete. For this animation, the stroke color changes from the original stroke color to `BLUE` during the first cycle and transitions back to the original stroke color during the second cycle.

`ParallelTransition`

Lines 42–43 configure a **`ParallelTransition`** that performs multiple transitions at the same time (that is, in parallel).

Click here to view code image

```
41          // parallelizes multiple transitions
42          var parallelTransition =
43             new ParallelTransition(fillTransition, strokeTransition);
44
```

The `ParallelTransition` constructor receives a variable number of `Transition`s as a comma-separated list. In this case, the `FillTransition` and `StrokeTransition` will be performed in parallel on the app's `Rectangle`.

FadeTransition

Lines 46–50 configure a one-second **FadeTransition** that changes a `Node`'s opacity.

Click here to view code image

```
45          // transition that changes a node's opacity over time
46          var fadeTransition = new FadeTransition(Duration.seconds(1));
47          fadeTransition.setFromValue(1.0); // opaque
48          fadeTransition.setToValue(0.0); // transparent
49          fadeTransition.setCycleCount(2);
50          fadeTransition.setAutoReverse(true);
51
```

Line 47 specifies the initial opacity—1.0 is fully opaque. Line 48 specifies the final opacity—0.0 is fully transparent. Once again, we set the cycle count to 2 and specified that the animation should auto-reverse itself.

RotateTransition

Lines 53–57 configure a one-second **RotateTransition** that rotates a `Node`.

Click here to view code image

```
52          // transition that rotates a node
53          var rotateTransition = new RotateTransition(Duration.seconds(1));
54          rotateTransition.setByAngle(360.0);
55          rotateTransition.setCycleCount(2);
56          rotateTransition.setInterpolator(Interpolator.EASE_BOTH);
57          rotateTransition.setAutoReverse(true);
58
```

You can rotate a `Node` by a specified number of degrees (line 54), or you can use other methods of class `RotateTransition` to specify a start angle and end angle. Each `Transition` animation uses an **`Interpolator`** to calculate new property values throughout the animation's duration. The default is a **`LINEAR`** `Interpolator`, which evenly divides the property value changes over the animation's duration. For the `RotateTransition`, line 56 uses the `Interpolator` **`EASE_BOTH`**, which changes the rotation slowly at first (known as "easing in"), speeds up the rotation in the middle of the animation, then slows the rotation again to complete the animation (known as "easing out"). For a list of `Interpolator`s, visit

Click here to view code image

```
https://openjfx.io/javadoc/23/javafx.graphics/javafx/animation/
   Interpolator.html
```

PathTransition

Lines 60–66 configure a two-second `PathTransition` that moves a node along a `Path`. The `Path` (lines 60–61) is specified as the second argument to the `PathTransition` constructor (lines 62–63). A `LineTo` object draws a straight line from the previous `PathElement`'s endpoint to the specified location. Line 65 specifies that this animation should use the `Interpolator` `EASE_IN`, which changes the position slowly at first before performing the animation at full speed.

Click here to view code image

```
59          // transition that moves a node along a Path
60          var path = new Path(new MoveTo(45, 45), new LineTo(45, 0),
61             new LineTo(90, 0), new LineTo(90, 90), new LineTo(0, 90));
62          var translateTransition =
63             new PathTransition(Duration.seconds(2), path);
64          translateTransition.setCycleCount(2);
65          translateTransition.setInterpolator(Interpolator.EASE_IN);
66          translateTransition.setAutoReverse(true);
67
```

ScaleTransition

Lines 69–74 configure a one-second `ScaleTransition` that changes a `Node`'s size. Lines 70–71 specify that the object will be scaled 75% larger along the x-axis (i.e., horizontally) and y-axis (i.e., vertically), respectively. Line 73 specifies that this animation should use the `Interpolator` `EASE_OUT`, which begins scaling the shape at full speed then slows down as the animation completes.

Click here to view code image

```
68          // transition that scales a shape to make it larger or smaller
69          var scaleTransition = new ScaleTransition(Duration.seconds(1));
70          scaleTransition.setByX(0.75);
71          scaleTransition.setByY(0.75);
72          scaleTransition.setCycleCount(2);
73          scaleTransition.setInterpolator(Interpolator.EASE_OUT);
74          scaleTransition.setAutoReverse(true);
75
```

SequentialTransition

Lines 77–79 configure a **SequentialTransition** that performs a sequence of transitions—as each completes, the next one begins executing. The `SequentialTransition` constructor receives the `Node` to which the sequence of animations will be applied, followed by a comma-separated list of `Transition`s to perform. In fact, every transition animation class has a constructor that enables you to specify a `Node`. For this example, we did not specify `Node`s when creating the other transitions because they're all applied to the `Rectangle` by the `SequentialTransition`. Every `Transition` has a **play** method (line 80) that begins the animation. Calling `play` on the `SequentialTransition` automatically calls `play` on each animation in the sequence.

Click here to view code image

```
76          // transition that applies a sequence of transitions to a node
77          var sequentialTransition = new SequentialTransition(
78             rectangle, parallelTransition, fadeTransition,
79             rotateTransition, translateTransition, scaleTransition);
80          sequentialTransition.play(); // play the transition
81       }
82    }
```

Generative AI

1 Prompt genAIs, asking why the terms "easing in" and "easing out" are used to describe animations that start slowly, then speed up, or start quickly, then slow down.

2 Prompt genAIs for a tutorial on the complete set of JavaFX `Interpolator`s with examples of how to use them.

3 Prompt genAIs, asking for the best practices to create smooth JavaFX `Transition` animations.

17.8 `Timeline` Animations

Next, we'll use a **Timeline** animation to bounce a `Circle` object around the app's `Pane` over time. A `Timeline` animation can change

any modifiable `Node` property. You specify how to change property values with one or more `KeyFrame` objects that the `Timeline` animation performs in sequence. For this app, we'll specify a single `KeyFrame` that modifies a `Circle`'s location, then we'll play that `KeyFrame` indefinitely.

`TimelineAnimation.fxml`

This app's FXML (Fig. 17.11) defines a `Circle` object (lines 13–15) with the `fx:id "c"`, a five-pixel black border and the fill color `DODGERBLUE`.

Click here to view code image

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!-- Fig. 17.11: TimelineAnimation.fxml -->
3  <!-- FXML for a Circle that will be animated by the controller -->
4
5  <?import javafx.scene.layout.Pane?>
6  <?import javafx.scene.shape.Circle?>
7
8  <Pane id="Pane" fx:id="pane" prefHeight="400.0"
9     prefWidth="600.0" xmlns:fx="http://javafx.com/fxml/1"
10    xmlns="http://javafx.com/javafx/23"
11    fx:controller="TimelineAnimationController">
12    <children>
13       <Circle fx:id="c" fill="DODGERBLUE" layoutX="142.0" layoutY="143.0"
14          radius="40.0" stroke="BLACK" strokeType="INSIDE"
15          strokeWidth="5.0" />
16    </children>
17 </Pane>
```

Fig. 17.11 | FXML for a `Circle` that will be animated by the controller.

`TimelineAnimationController.java`

Class `TimelineAnimationController`'s (Fig. 17.12) `initialize` method (lines 20–43) configures then plays the `Timeline` animation. Lines 23–42 define the animation, line 41 specifies that the animation should cycle indefinitely (until the animation's `stop` method is called), and line 42 plays the animation.

Click here to view code image

```
1  // Fig. 17.12: TimelineAnimationController.java
2  // Bounce a circle around a window using a Timeline animation
```

```
3  import java.util.random.RandomGenerator;
4  import javafx.animation.KeyFrame;
5  import javafx.animation.Timeline;
6  import javafx.fxml.FXML;
7  import javafx.geometry.Bounds;
8  import javafx.scene.layout.Pane;
9  import javafx.scene.shape.Circle;
10 import javafx.util.Duration;
11
12 public class TimelineAnimationController {
13    private final static RandomGenerator random =
14       RandomGenerator.getDefault();
15    private int dx = random.nextInt(1, 6);
16    private int dy = random.nextInt(1, 6);
17    @FXML private Circle c;
18    @FXML private Pane pane;
19
20    @FXML
21    private void initialize() {
22       // define a timeline animation
23       var timelineAnimation = new Timeline(
24          new KeyFrame(Duration.millis(10), e -> {
25             // move the circle by the dx and dy amounts
26             c.setLayoutX(c.getLayoutX() + dx);
27             c.setLayoutY(c.getLayoutY() + dy);
28             Bounds bounds = pane.getBoundsInLocal();
29
30             if (hitRightOrLeftEdge(bounds)) {
31                dx *= -1;
32             }
33
34             if (hitTopOrBottom(bounds)) {
35                dy *= -1;
36             }
37          }) // end call to KeyFrame constructor
38       );
39
40       // indicate that the timeline animation should run indefinitely
41       timelineAnimation.setCycleCount(Timeline.INDEFINITE);
42       timelineAnimation.play();
43    }
44
45    // determines whether the circle hit the left or right of the window
46    private boolean hitRightOrLeftEdge(Bounds bounds) {
47       return (c.getLayoutX() <= (bounds.getMinX() + c.getRadius())) ||
48          (c.getLayoutX() >= (bounds.getMaxX() - c.getRadius()));
49       }
50
51    // determines whether the circle hit the top or bottom of the window
52    private boolean hitTopOrBottom(Bounds bounds) {
```

```
53          return (c.getLayoutY() <= (bounds.getMinY() + c.getRadius())) ||
54             (c.getLayoutY() >= (bounds.getMaxY() - c.getRadius()));
55       }
56    }
```

Fig. 17.12 | Bounce a circle around a window using a `Timeline` animation.

Creating the `Timeline`

The `Timeline` constructor used in lines 23–38 can receive a comma-separated list of `KeyFrame`s as arguments—in this case, we pass one. Each `KeyFrame` issues an `ActionEvent` at a particular time in the animation. The app can respond to the event by changing a `Node`'s property values. The `KeyFrame` constructor used here specifies that, after ten milliseconds, the `ActionEvent` will occur. We set the `Timeline`'s cycle count to `Timeline.INDEFINITE`, so the `Timeline` will perform this `KeyFrame` every ten milliseconds. Lines 24–37 use a lambda to define the `EventHandler` for the `KeyFrame`'s `ActionEvent`. The lambda's parameter (`e`) receives the `ActionEvent`.

`KeyFrame`'s `EventHandler`

The lambda uses class `TimelineAnimationController`'s instance variables `dx` and `dy` (declared and initialized at lines 15–16) to change the `Circle`'s x- and y-coordinates each time the `KeyFrame` plays. Lines 26–27 add these values to the `Circle`'s x- and y-coordinates. Next, lines 30–36 determine whether the `Circle` has collided with any of the `Pane`'s edges. If the `Circle` hits the left or right edge, line 31 multiplies `dx`'s value by `-1` to reverse the `Circle`'s horizontal direction. If the `Circle` hits the top or bottom edge, line 35 multiplies `dy`'s value by `-1` to reverse the `Circle`'s horizontal direction.

Generative AI

1 Prompt genAIs, asking what the best practices are for creating smooth JavaFX `Transition` animations.

2 Prompt genAIs, asking for examples of why you might combine JavaFX `Timeline` and `Transition` animations.

17.9 Frame-by-Frame Animation with `AnimationTimer`

A third way to implement JavaFX animations is via the `javafx.animation` package's **class `AnimationTimer`**, which enables you to define frame-by-frame animations. You specify how your objects should move in a given frame. JavaFX aggregates all of the drawing operations and displays the frame. This can be used with objects in the scene graph or to draw shapes in a `Canvas`. JavaFX calls the **`handle`** method of every `AnimationTimer` before it draws an animation frame.

JavaFX tries to display animation frames at 60 frames per second for smooth animation. This frame rate varies based on the animation's complexity, the processor speed and how busy the processor is at a given time. For this reason, method `handle` receives a time stamp in nanoseconds (billionths of a second) that you can use to determine the elapsed time since the last animation frame, then you can scale the movements of your objects accordingly. This enables you to define animations that operate at the same overall speed, regardless of the frame rate on a given device. We'll discuss JavaFX performance issues in more detail in Chapter 18.

Figure 17.13 reimplements the animation in Fig. 17.12 using an `AnimationTimer`. The FXML is identical (other than the filename and controller class name). Much of the code is identical to Fig. 17.12, so we focus on the new features.

Click here to view code image

```
1  // Fig. 17.13: BallAnimationTimerController.java
2  // Bounce a circle around a window using an AnimationTimer subclass.
3  import java.util.random.RandomGenerator;
4  import javafx.animation.AnimationTimer;
5  import javafx.fxml.FXML;
6  import javafx.geometry.Bounds;
7  import javafx.scene.layout.Pane;
8  import javafx.scene.shape.Circle;
9
10 public class BallAnimationTimerController {
11    private final static RandomGenerator random =
12       RandomGenerator.getDefault();
13    private int dx = random.nextInt(1, 6);
```

```
14      private int dy = random.nextInt(1, 6);
15      @FXML private Circle c;
16      @FXML private Pane pane;
17
18      @FXML
19      private void initialize() {
20         // define an AnimationTimer
21         AnimationTimer timer = new AnimationTimer() {
22            int velocity = 60; // used to scale distance changes
23            long previousTime = System.nanoTime(); // time since app launch
24
25            // specify how to move Circle for current animation frame
26            @Override
27            public void handle(long now) {
28               double elapsedTime = (now - previousTime) / 1000000000.0;
29               previousTime = now;
30               double scale = elapsedTime * velocity;
31
32               Bounds bounds = pane.getBoundsInLocal();
33               c.setLayoutX(c.getLayoutX() + dx * scale);
34               c.setLayoutY(c.getLayoutY() + dy * scale);
35
36               if (hitRightOrLeftEdge(bounds)) {
37                  dx *= -1;
38               }
39
40               if (hitTopOrBottom(bounds)) {
41                  dy *= -1;
42               }
43            }
44         };
45
46         timer.start();
47      }
48
49      // determines whether the circle hit left/right of the window
50      private boolean hitRightOrLeftEdge(Bounds bounds) {
51         return (c.getLayoutX() <= (bounds.getMinX() + c.getRadius())) ||
52            (c.getLayoutX() >= (bounds.getMaxX() - c.getRadius()));
53      }
54
55      // determines whether the circle hit top/bottom of the window
56      private boolean hitTopOrBottom(Bounds bounds) {
57         return (c.getLayoutY() <= (bounds.getMinY() + c.getRadius())) ||
58            (c.getLayoutY() >= (bounds.getMaxY() - c.getRadius()));
59      }
60   }
```

Fig. 17.13 | Bounce a circle around a window using an `AnimationTimer` subclass.

Extending `abstract` Class `AnimationTimer` Using an Anonymous Inner Class

Class `AnimationTimer` is an `abstract` class, so you must create a subclass. In this example, rather than defining an `AnimationTimer` subclass in a separate file, lines 21–44 create an **anonymous inner class**. Such a class does not have a name and is typically declared in a method. Lines 21–44 are one statement that

- declares the `AnimationTimer` subclass,
- immediately creates an object of that class and
- assigns the new object's reference to the `AnimationTimer` variable `timer` (line 21).

Since an anonymous inner class has no name, you must create an object of the class at the point where it's declared, thus the keyword `new` in line 21. The parentheses in

```
new AnimationTimer()
```

call the anonymous class's constructor. After the parentheses are the braces that delimit the anonymous class's body (lines 21 and 44), which may contain field and method declarations like other classes. Lines 22–23 define the anonymous inner class's instance variables:

- Variable `velocity` is used as a multiplier to determine the distance moved in each animation frame. We discuss this momentarily.
- Variable `previousTime` represents the time stamp (in nanoseconds) of the previous animation frame—this will be used to determine the elapsed time between frames. We initialized `previousTime` to `System.nanoTime()`, which returns the number of nanoseconds since the JVM launched the app. `AnimationTimer`'s `handle` method also receives as its argument the number of nanoseconds since the JVM launched the app.

Overriding `AnimationTimer`'s `handle` Method

Lines 26–43 override class `AnimationTimer`'s `handle` method, which specifies what to do during each animation frame:

- Line 28 calculates the `elapsedTime` in seconds since the last

animation frame. If method `handle` is called 60 times per second, the elapsed time between frames will be approximately 0.0167 seconds—that is, 1/60 of a second.

- Line 29 stores the time stamp in `previousTime` for use in the next animation frame.
- When we change the `Circle`'s `layoutX` and `layoutY` (lines 33–34), we multiply `dx` and `dy` by the `scale` (line 30). In Fig. 17.12, the `Circle`'s speed was determined by moving between one and five pixels along the x- and y-axes every 10 milliseconds—the larger the values, the faster the `Circle` moved. If we were to scale `dx` or `dy` by just `elapsedTime`, we'd move the `Circle` only small fractions of `dx` and `dy` during each frame—approximately 0.0167 seconds (1/60 of a second) to 0.083 seconds (5/60 of a second), based on their randomly chosen values. For this reason, we multiply the `elapsedTime` by the `velocity` (60) to scale the movement in each frame. This results in values that are approximately one to five pixels, as in Fig. 17.12.

After creating the `AnimationTimer` object, the controller class's `initialize` method calls the `AnimationTimer` object's `start` method (line 46) to begin animating the ball.

Anonymous Inner Class Notes

An anonymous inner class can access its top-level class's instance variables, `static` variables and methods. In this example, the anonymous inner class uses the controller class's instance variables `dx` and `dy`. An anonymous inner class has limited access to the local variables of the method in which it's declared. It can access only the `final` or effectively `final` local variables declared in the enclosing method's body.

Generative AI

1 Prompt genAIs to discuss the differences between Java's regular classes and anonymous inner classes.

2 Prompt genAIs to discuss the differences between JavaFX's `Timeline`-based animations and `AnimationTimer`-based animations and when to use each.

17.10 CSS Transitions

Sections 17.2–17.3 introduced JavaFX CSS for styling JavaFX GUIs. JavaFX's CSS support traditionally has focused on static `Node` styling. If you wanted to animate a `Node` property, you'd use the programmatically applied JavaFX transitions shown in Section 17.7. In web development, **CSS transitions** are widely used to create smooth animation effects. These are often applied in response to the user **hovering** the mouse cursor over a webpage element—that is, moving the mouse cursor into a webpage element's bounds. When the user moves the mouse cursor outside the element's bounds, the element typically returns to its original appearance. Some CSS hover transitions include:[7][8][9]

7. "CSS Transitions." Accessed October 3, 2024. https://www.w3.org/TR/css-transitions-1/.
8. "Using CSS Transitions." Accessed October 3, 2024. https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_transitions/Using_CSS_transitions.
9. "Transitions." Accessed October 3, 2024. https://web.dev/learn/css/transitions.

- changing an element's appearance to indicate it's clickable,
- scaling an element's size (zooming) to draw attention to it,
- changing an image (such as an icon) temporarily to show that an element is interactive,
- highlighting menu items as the user moves the mouse cursor over them,
- moving elements to grab a user's attention,
- changing an element's rotation angle,
- adding shadows to make an element appear as if it's raised off the screen, and
- fading an element in so it's visible or out so it's not visible.

Other events that can trigger transitions include when a control receives the focus or is pressed.

CSS Transitions in JavaFX 23

JavaFX 23 adds support for webpage-style CSS transitions,[10] which you can apply to any `Node` property styled with JavaFX CSS properties. The CSS transitions execute automatically in response to events, like hovering the mouse cursor over a `Node`. For precise animation control, you can use the other animation techniques shown in Sections 17.7–17.9.

10. "JavaFX CSS Reference Guide—Transitions." Accessed October 3, 2024. https://openjfx.io/javadoc/23/javafx.graphics/javafx/scene/doc-files/cssref.html#introtransitions.

CSSTransition.fxml

To demonstrate CSS transitions, we'll use an FXML GUI consisting of a `Pane` layout containing a rounded `Rectangle` (Fig. 17.14). Line 9 references the CSS stylesheet containing the `Rectangle`'s properties and the CSS transitions (presented in Fig. 17.15).

Click here to view code image

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!-- Fig. 17.14: CSSTransition.fxml -->
3  <!-- FXML for a Pane and a Rectangle -->
4
5  <?import javafx.scene.layout.Pane?>
6  <?import javafx.scene.shape.Rectangle?>
7
8  <Pane prefHeight="150.0" prefWidth="150.0"
9     stylesheets="@CSSTransition.css"
10    xmlns="http://javafx.com/javafx/23"
11    xmlns:fx="http://javafx.com/fxml/1">
12    <children>
13       <Rectangle fx:id="rectangle" height="90.0"
14          layoutX="30.0" layoutY="30.0" width="90.0" />
15    </children>
16 </Pane>
```

Fig. 17.14 | FXML for a `Pane` and a `Rectangle`.

TransitionAnimations.fxml

Using CSS transitions, we'll modify the `Rectangle`'s fill, stroke color, stroke width and rotation angle. The following screen captures show the transition progressions—they begin when the user hovers the mouse cursor over the `Rectangle` and transition back to the original state when the user moves the mouse outside the `Rectangle`'s bounds:

CSSTransition.css

Figure 17.15 contains this example's CSS stylesheet. In the `Rectangle` CSS rule:

- lines 4–9 specify the `Rectangle`'s initial styling, and
- lines 10–12 specify the CSS transition settings (discussed momentarily).

The **`-fx-rotate` property** specifies the `Rectangle`'s rotation angle in degrees. The CSS rule in lines 15–20 specifies the values of the four CSS properties JavaFX will apply when the user hovers the mouse cursor over the `Rectangle`. The notation **`:hover`** is a **CSS pseudo-class**—JavaFX applies this rule only when the mouse cursor hovers over the `Node` that matches the rule.[11] The CSS properties return to their original values in lines 4, 5, 8 and 9 when the user moves the mouse out of the `Rectangle`.

11. Visit the "Pseudo-classes" section of the JavaFX CSS reference at `https://openjfx.io/javadoc/23/javafx.graphics/javafx/scene/doc-files/cssref.html` for other pseudo-classes that can trigger transitions.

Click here to view code image

```
1  /* Fig. 17.15: CSSTransition.css */
2  /* CSS rules to apply styling and CSS transitions to a Rectangle. */
3  Rectangle {
4     -fx-stroke-width: 5;
5     -fx-stroke: black;
6     -fx-arc-width: 50;
7     -fx-arc-height: 50;
8     -fx-fill: red;
9     -fx-rotate: 0;
10    transition-property: -fx-fill, -fx-stroke,
11       -fx-stroke-width, -fx-rotate;
12    transition-duration: 3.0s;
13 }
14
15 Rectangle:hover {
16    -fx-stroke-width: 15;
17    -fx-stroke: red;
18    -fx-fill: black;
19    -fx-rotate: 180;
20 }
```

Fig. 17.15 | CSS rules to apply styling and CSS transitions to a `Rectangle`.

Specifying the Transition Settings

The **CSS `transition-property`** (lines 10–11) specifies the `Rectangle` rule's CSS properties that will participate in the CSS transition animation—in this case:

- `-fx-fill` (the `Rectangle`'s fill color),
- `-fx-stroke` (the `Rectangle`'s border color),
- `-fx-stroke-width` (the `Rectangle`'s border thickness) and
- `-fx-rotate` (the `Node`'s rotation angle in degrees).

The **CSS `transition-duration`** (line 12) specifies the transition animation's duration (`3s` indicates three seconds).

When the user hovers the mouse over the `Rectangle`, JavaFX automatically applies a smooth animation for the properties listed in the `transition-property`. As with the `Transition` animations shown in Section 17.7, CSS animations interpolate between the properties' original values (lines 4, 5, 8 and 9) and their new values (lines 16–19). All the CSS transitions for the properties listed in the `transition-property` (lines 10–11) are applied to the `Rectangle` in parallel. When the user moves the mouse cursor out of `Rectangle`'s bounds, JavaFX automatically applies smooth transition animations that return each property to its original value (lines 4, 5, 8 and 9).

Other JavaFX CSS Transition Properties

Each CSS transition has an associated **`transition-timing-function`**. By default, CSS transitions use the **`ease` timing function**, which begins the transition slowly ("easing in"), speeds up the transition in the middle then slows the transition at the end ("easing out")—like `Interpolator.EASE_BOTH` in Fig. 17.10. You can customize the `transition-timing-function` by adding the property

Click here to view code image

```
transition-timing-function: value;
```

to the CSS rule that specifies the `transition-property`. For the list of timing functions, visit:

Click here to view code image

```
https://openjfx.io/javadoc/23/javafx.graphics/javafx/scene/doc-
   files/cssref.html#typeeasingfunction
```

You may also delay the transition animation's start by a specified time interval using the `transition-delay` **property**.

You can combine all the CSS transition properties into a single shorthand `transition` **property** of the form:

Click here to view code image

```
transition:
   property duration timing-function delay,
   …
   property duration timing-function delay;
```

in which you specify the *property*, *duration*, *timing-function* and *delay* for each CSS property that participates in the CSS transition. For example, we could write this example's `transition` property as:

Click here to view code image

```
transition:
   -fx-fill 3.0s ease 0s,
   -fx-stroke 3.0s ease 0s,
   -fx-stroke-width 3.0s ease 0s,
   -fx-rotate 3.0s ease 0s;
```

The *timing-function* and *delay* shown here are the defaults, so we can exclude them, as in:

```
transition:
   -fx-fill 3.0s,
   -fx-stroke 3.0s,
   -fx-stroke-width 3.0s,
   -fx-rotate 3.0s;
```

JavaFX CSS Transition Notes

There are several cases in which JavaFX CSS transitions are ignored or canceled:[12]

12. "JavaFX CSS Reference Guide—Transitions." Accessed October 3, 2024. https://openjfx.io/javadoc/23/javafx.graphics/javafx/scene/doc-files/cssref.html#introtransitions.

- your program sets the corresponding `Node` property programmatically,
- the property is used in a data binding,

- the corresponding node is not visible on the screen or
- the node has been removed from the scene.

CSS Transition Resources

At the time of this writing, JavaFX CSS transitions are new, and there is little online documentation about them. However, they're based directly on the CSS transitions used in web development. By studying discussions of the transitions used in web development, you can get ideas for how JavaFX CSS properties can be animated. In particular, the **CSS Transition Generator** at

Click here to view code image

```
https://dev.to/smpnjn/css-transition-generator-5de0
```

enables you to experiment with various transition settings and view the corresponding CSS. You can copy the settings into your JavaFX CSS stylesheets to apply the settings to JavaFX `Node`s with similar CSS properties. For additional CSS transition information, visit:

- https://www.w3.org/TR/css-transitions-1/
- https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_transitions/Using_CSS_transitions
- https://www.w3schools.com/css/css3_transitions.asp

Generative AI

1 Prompt genAIs, asking them to discuss the pros and cons of using JavaFX 23's CSS transitions vs. JavaFX's `Transition` animations and when to prefer each.

2 Prompt genAIs, asking the recommended duration for CSS hover animations.

3 Prompt genAIs to suggest cool CSS transitions used in webpage development that could be applied to JavaFX `Node`s.

17.11 Drawing on a `Canvas`

So far, you've displayed and manipulated JavaFX two-dimensional shape objects in the JavaFX scene graph. This section demonstrates similar drawing capabilities using the `javafx.scene.canvas` package, which contains two classes:

- Class `Canvas` is a subclass of `Node` in which you can draw graphics.
- **Class `GraphicsContext`** performs drawing operations on a `Canvas`.

A `GraphicsContext` object supports the same drawing characteristics you've previously used with `Shape`s, but you set these characteristics, then draw the shapes programmatically. Figure 17.16 re-implements Section 17.3's `BasicShapes` example to demonstrate `Canvas` capabilities, producing the following output:

You'll use various JavaFX classes and `enum`s from packages `javafx.scene.image`, `javafx.scene.paint` and `javafx.scene.shape`. JavaFX's CSS capabilities use these behind the scenes to style `Shape`s. A `Canvas` is typically preferred for performance-oriented graphics, such as games with moving elements. We split the code (and specifically method `initialize`) into parts for discussion purposes.

Click here to view code image

```
1  // Fig. 17.16: CanvasShapesController.java
2  // Drawing on a Canvas.
3  import javafx.fxml.FXML;
4  import javafx.scene.canvas.Canvas;
5  import javafx.scene.canvas.GraphicsContext;
6  import javafx.scene.image.Image;
7  import javafx.scene.paint.Color;
8  import javafx.scene.paint.CycleMethod;
9  import javafx.scene.paint.ImagePattern;
10 import javafx.scene.paint.LinearGradient;
11 import javafx.scene.paint.RadialGradient;
12 import javafx.scene.paint.Stop;
13 import javafx.scene.shape.ArcType;
14 import javafx.scene.shape.StrokeLineCap;
15
16 public class CanvasShapesController {
17    // instance variables that refer to GUI components
18    @FXML private Canvas drawingCanvas;
19
```

Fig. 17.16 | Drawing on a `Canvas`.

Obtaining the `GraphicsContext` and Setting the Line Width for All Shapes

To draw on a `Canvas`, you first obtain its `GraphicsContext` by calling `Canvas` method **`getGraphicsContext2D`** (line 23).

Click here to view code image

```
20      // draw on the Canvas
21      @FXML
22      private void initialize() {
23         GraphicsContext gc = drawingCanvas.getGraphicsContext2D();
24         gc.setLineWidth(10); // set all stroke widths
25
```

When you set a `GraphicsContext`'s drawing characteristics, they're applied (as appropriate) to all subsequent shapes you draw. For example, line 24 calls the **`GraphicsContext` method `setLineWidth`** to specify the line thickness (`10`). All subsequent `GraphicsContext` method calls that draw lines or shape borders will use this line-width setting. This is similar to the `-fx-stroke-width` CSS attribute we specified for all shapes in Section 17.3.

Drawing Lines

Lines 27–34 draw the red and green lines:

Click here to view code image

```
26         // draw red line
27         gc.setStroke(Color.RED);
28         gc.strokeLine(10, 10, 100, 100);
29
30         // draw green line
31         gc.setGlobalAlpha(0.5); // half transparent
32         gc.setLineCap(StrokeLineCap.ROUND);
33         gc.setStroke(Color.GREEN);
34         gc.strokeLine(100, 10, 10, 100);
35
36         gc.setGlobalAlpha(1.0); // reset alpha transparency
37
```

- **`GraphicsContext`'s `setStroke` method** (lines 27 and 33) specifies the **`Paint`** object (package `javafx.scene.paint`) used to draw the line. This can be any of the `Paint` subclasses **`Color`**, **`ImagePattern`**, **`LinearGradient`** or **`RadialGradient`** (all from package `javafx.scene.paint`).

- **`GraphicsContext`'s `strokeLine` method** (lines 28 and 34) draws a line using the current `Paint` object we set as the stroke. The four arguments are the *x*-*y* coordinates of the start and end points.
- **`GraphicsContext`'s `setGlobalAlpha` method** (line 31) sets the alpha transparency of all subsequent shapes you draw. For the green line, we used `0.5`, which is 50% transparent. After drawing the green line, we reset this to the default `1.0` (line 35) so subsequent shapes are fully opaque.
- **`GraphicsContext`'s `setLineCap` method** (line 32) sets the line cap, like the CSS property `-fx-stroke-line-cap` in Section 17.3. This method's argument is a constant from the **`enum StrokeLineCap`** (package `javafx.scene.shape`). Here, we round the line ends.

Drawing a Rounded Rectangle

Lines 39–42 draw a rounded rectangle with a red border and yellow fill:

Click here to view code image

```
38          // draw rounded rect with red border and yellow fill
39          gc.setStroke(Color.RED);
40          gc.setFill(Color.YELLOW);
41          gc.fillRoundRect(120, 10, 90, 90, 50, 50);
42          gc.strokeRoundRect(120, 10, 90, 90, 50, 50);
43
```

- Line 38 sets the border color to `Color.RED`.
- **`GraphicsContext`'s `setFill` method** (lines 40, 50, 56 and 66) specifies the `Paint` object that fills a shape. Here we fill the rectangle with `Color.YELLOW`.
- **`GraphicsContext`'s `fillRoundRect` method** (line 41) draws a filled rectangle with rounded corners using the current `Paint` object as the fill. The first four arguments represent the rectangle's upper-left *x*-coordinate, upper-left *y*-coordinate, width and height. The last two arguments specify the arc width and arc height for rounding the corners. These work identically to the CSS properties `-fx-arc-width` and `-fx-arc-height` properties in Section 17.3. `GraphicsContext` also provides a **`fillRect` method** that draws a rectangle without

rounded corners.

- **`GraphicsContext`'s `strokeRoundRect` method** (line 42) has the same arguments as method `fillRoundRect` but draws a hollow rectangle with rounded corners. `GraphicsContext` also provides a **`strokeRect` method** that draws a rectangle without rounded corners.

Drawing a Circle with a `RadialGradient` Fill

Lines 45–52 draw a circle with a blue border and a red-white, radial-gradient fill:

Click here to view code image

```
44          // draw circle with blue border and red/white radial gradient fill
45          gc.setStroke(Color.BLUE);
46          Stop[] stopsRadial =
47             {new Stop(0, Color.RED), new Stop(1, Color.WHITE)};
48          var radialGradient = new RadialGradient(0, 0, 0.5, 0.5,
49             0.6, true, CycleMethod.NO_CYCLE, stopsRadial);
50          gc.setFill(radialGradient);
51          gc.fillOval(230, 10, 90, 90);
52          gc.strokeOval(230, 10, 90, 90);
53
```

Line 45 sets the border color to `Color.BLUE`. Lines 46–49 configure the `RadialGradient`—these lines perform the same tasks as the CSS function `radial-gradient` in Section 17.3. Lines 46–47 create an array of **`Stop`** objects (package `javafx.scene.paint`) representing the color stops. Each has an offset from 0.0 to 1.0 (a percentage from the gradient's start point) and a `Color`. Here, the `Stop`s indicate that the radial gradient will transition from red at the gradient's start point to white at its endpoint.

The `RadialGradient` constructor call in lines 48–49 creates a red-white radial gradient that starts with solid red at the shape's center and transitions to white at 60% of the radial gradient's radius. The constructor's arguments are:

- the focus angle, which specifies the direction of the radial gradient's focal point from the gradient's center,
- the distance of the focal point as a percentage (0.0–1.0)—using 0 for both the focus angle and focal point indicates the gradient should begin from the oval's center,
- the center point's x and y location as percentages (0.0–1.0)

of the width and height for the shape being filled,

- a `boolean` indicating whether the gradient should scale to fill its shape,
- a constant from the **`CycleMethod enum`** (package `javafx.scene.paint`) indicating how the color stops are applied and
- an array of `Stop` objects—this can also be a comma-separated list of `Stop`s or a `List<Stop>` object.

Line 50 sets the fill to the new `radialGradient`, then lines 51–52 call **`GraphicsContext`**'s **`fillOval`** and **`strokeOval`** **methods** to draw a filled oval and a hollow oval, respectively. Each method receives as arguments the upper-left x-coordinate, upper-left y-coordinate, width and height of the rectangular area in which the oval should be drawn—known as a bounding box. Because the width and height are the same, these calls draw circles.

Drawing an Oval with an `ImagePattern` Fill

Lines 55–58 draw an oval with a green border and containing an image:

Click here to view code image

```
54          // draw ellipse with green border and image fill
55          gc.setStroke(Color.GREEN);
56          gc.setFill(new ImagePattern(new Image("yellowflowers.png")));
57          gc.fillOval(340, 10, 200, 90);
58          gc.strokeOval(340, 10, 200, 90);
59
```

- Line 55 sets the border color to `Color.GREEN`.
- Line 56 sets the fill to an `ImagePattern`—a subclass of `Paint` that loads an `Image` from the local system or from a URL specified as a `String`. `ImagePattern` is the class used by the CSS function `image-pattern` in Section 17.3.
- Lines 57–58 draw a filled and a hollow oval.

Drawing an Arc with a `LinearGradient` Fill

Lines 61–68 draw an arc with a purple border and filled with a cyan-white linear gradient:

Click here to view code image

```
60          // draw arc with purple border and cyan/white linear gradient fill
61          gc.setStroke(Color.PURPLE);
62          Stop[] stopsLinear =
63             {new Stop(0, Color.CYAN), new Stop(1, Color.WHITE)};
64          var linearGradient = new LinearGradient(0, 0, 1, 0,
65             true, CycleMethod.NO_CYCLE, stopsLinear);
66          gc.setFill(linearGradient);
67          gc.fillArc(560, 10, 90, 90, 45, 270, ArcType.ROUND);
68          gc.strokeArc(560, 10, 90, 90, 45, 270, ArcType.ROUND);
69       }
70    }
```

- Line 61 sets the border color to `Color.PURPLE`.
- Lines 62–65 configure the `LinearGradient`—the class used by the CSS function `linear-gradient` in Section 17.3. The constructor's first four arguments are the endpoint coordinates that represent the direction and angle of the gradient. If the x-coordinates are the same, the gradient is vertical. If the y-coordinates are the same, the gradient is horizontal. All other linear gradients follow a diagonal line. When these values are specified in the range 0.0 to 1.0, and the constructor's fifth argument is `true`, the gradient is scaled to fill the shape. The next argument is the `CycleMethod`. The last argument is an array of `Stop` objects—again, this can be a comma-separated list of `Stop`s or a `List<Stop>` object.

Lines 67–68 call **`GraphicsContext`'s `fillArc` and `strokeArc` methods** to draw a filled arc and a hollow arc, respectively. Each method receives as arguments:

- the upper-left x-coordinate, upper-left y-coordinate, width and height of the rectangular area (that is, the bounding box) in which the oval should be drawn,
- the start angle and sweep of the arc in degrees, and
- a constant from the **`ArcType` `enum`** (package `javafx.scene.shape`)

Additional `GraphicsContext` Features

There are many additional `GraphicsContext` features, which you

can explore at

Click here to view code image

```
https://openjfx.io/javadoc/23/javafx.graphics/javafx/scene/canvas/
   GraphicsContext.html
```

Some of the capabilities that we did not discuss here include:

- Drawing and filling text—similar to the font features in Section 17.2.
- Drawing and filling polylines, polygons and paths—similar to the corresponding `Shape` subclasses in Section 17.4.
- Applying effects and transforms—similar to the transforms in Section 17.5.
- Drawing images.
- Manipulating the individual pixels of a drawing in a `Canvas` via a `PixelWriter`.
- Saving and restoring graphics characteristics via the `save` and `restore` methods.

Generative AI

1 Prompt genAIs to discuss when to use JavaFX's `Canvas` vs. the scene graph, including their differences with regard to performance, event handling and CSS styling.

2 Prompt genAIs to discuss a `RadialGradient`'s focus angle and focal point constructor arguments in detail.

17.12 Three-Dimensional Shapes

We've demonstrated many two-dimensional graphics capabilities. JavaFX provides several three-dimensional shapes and corresponding capabilities. The three-dimensional shapes are subclasses of `Shape3D` from the package `javafx.scene.shape`. In this section, you'll use Scene Builder to create a **`Box`**, a **`Cylinder`** and a **`Sphere`** and specify several of their properties. Then, in the app's controller, you'll create materials that apply color and images

to the 3D shapes.[13]

13. At the time of this writing, 3D shapes are not rendering correctly on M-processor (ARM) Macs in Scene Builder or in the running app, and 3D shapes are not rendering at all on Windows 11 ARM. We'll post updates to the book's web page: `https://deitel.com/javafp5`.

FXML for the `Box`, `Cylinder` and `Sphere`

Figure 17.17 presents the completed FXML that we created with Scene Builder:

- Lines 16–21 define the `Box` object.
- Lines 22–27 define the `Cylinder` object.
- Lines 28–29 define the `Sphere` object.

We dragged objects of these `Shape3D` subclasses from the Scene Builder Library's Shapes section onto the design area and gave them the fx:id values `box`, `cylinder` and `sphere`, respectively. We also set the controller to `ThreeDimensionalShapesController`.

Click here to view code image

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!-- Fig. 17.17: ThreeDimensionalShapes.fxml -->
3  <!-- FXML that displays a Box, Cylinder and Sphere -->
4
5  <?import javafx.geometry.Point3D?>
6  <?import javafx.scene.layout.Pane?>
7  <?import javafx.scene.shape.Box?>
8  <?import javafx.scene.shape.Cylinder?>
9  <?import javafx.scene.shape.Sphere?>
10
11 <Pane prefHeight="200.0" prefWidth="510.0"
12    xmlns="http://javafx.com/javafx/23"
13    xmlns:fx="http://javafx.com/fxml/1"
14    fx:controller="ThreeDimensionalShapesController">
15    <children>
16       <Box fx:id="box" depth="100.0" height="100.0" layoutX="100.0"
17          layoutY="100.0" rotate="30.0" width="100.0">
18          <rotationAxis>
19             <Point3D x="1.0" y="1.0" z="1.0" />
20          </rotationAxis>
21       </Box>
22       <Cylinder fx:id="cylinder" height="100.0" layoutX="265.0"
23          layoutY="100.0" radius="50.0" rotate="-45.0">
24          <rotationAxis>
25             <Point3D x="1.0" y="1.0" z="1.0" />
```

```
            </rotationAxis>
         </Cylinder>
         <Sphere fx:id="sphere" layoutX="430.0" layoutY="100.0"
            radius="60.0" />
      </children>
   </Pane>
```

Fig. 17.17 | FXML that displays a `Box`, `Cylinder` and `Sphere`.

All three shapes are initially gray. The shading in Scene Builder comes from the scene's default lighting. Though we do not use them here, package `javafx.scene`'s classes `AmbientLight`, `PointLight`, `SpotLight` and `DirectionalLight` can be used to add lighting effects. You can also use camera objects to view the scene from different angles and distances. The `AmbientLight`, `PointLight` and camera capabilities are located in the Scene Builder Library's 3D section.

`Box` Properties

Configure the `Box`'s properties in the Scene Builder Inspector's Layout section as follows:

- Set Width, Height and Depth to `100`, making a cube. The depth is measured along the z-axis, which runs perpendicular to your screen. When you move objects along the z-axis, they get bigger as they're brought toward you and smaller as they're moved away from you.
- Set Layout X and Layout Y to `100` to specify the cube's location.
- Set Rotate to `30` to specify the rotation angle in degrees. Positive values rotate counter-clockwise.
- For Rotation Axis, set the X, Y and Z values to `1` to indicate that the Rotate angle should be used to rotate the cube 30 degrees around each axis.

To observe how the Rotate angle and Rotation Axis values affect the `Box`'s rotation, set two of the three Rotation Axis values to `0`, then change the Rotate angle.

`Cylinder` Properties

Configure the `Cylinder`'s properties in Scene Builder as follows:

- Set Height to `100.0` and Radius to `50`.
- Set Layout X and Layout Y to `265` and `100`, respectively.
- Set Rotate to `-45` to specify the rotation angle in degrees. Negative values rotate clockwise.
- For Rotation Axis, set the X, Y and Z values to `1` to indicate that the Rotate angle should be applied to all three axes.

`Sphere` Properties

Configure the `Sphere`'s properties in Scene Builder as follows:

- Set Radius to `60`.
- Set Layout X and Layout Y to `430` and `100`, respectively.

`ThreeDimensionalShapesController` Class

Figure 17.18 presents this app's controller and final output. The final shapes' colors and images are created by applying so-called materials to the shapes. A **`PhongMaterial`** (package `javafx.scene.paint`) defines materials for 3D shapes. The name "Phong" is a 3D graphics term. Phong shading is a technique for applying color and shading to 3D surfaces. For more details on this technique, visit

Click here to view code image

```
https://en.wikipedia.org/wiki/Phong_shading
```

Click here to view code image

```
1  // Fig. 17.18: ThreeDimensionalShapesController.java
2  // Setting the material displayed on 3D shapes.
3  import javafx.fxml.FXML;
4  import javafx.scene.paint.Color;
5  import javafx.scene.paint.PhongMaterial;
6  import javafx.scene.image.Image;
7  import javafx.scene.shape.Box;
8  import javafx.scene.shape.Cylinder;
9  import javafx.scene.shape.Sphere;
10
11 public class ThreeDimensionalShapesController {
12    // instance variables that refer to 3D shapes
13    @FXML private Box box;
14    @FXML private Cylinder cylinder;
15    @FXML private Sphere sphere;
```

```
16
17      // set the material for each 3D shape
18      @FXML
19      private void initialize() {
20         // define material for the Box object
21         var boxMaterial = new PhongMaterial();
22         boxMaterial.setDiffuseColor(Color.CYAN);
23         box.setMaterial(boxMaterial);
24
25         // define material for the Cylinder object
26         var cylinderMaterial = new PhongMaterial();
27         cylinderMaterial.setDiffuseMap(new Image("yellowflowers.png"));
28         cylinder.setMaterial(cylinderMaterial);
29
30         // define material for the Sphere object
31         var sphereMaterial = new PhongMaterial();
32         sphereMaterial.setDiffuseColor(Color.RED);
33         sphereMaterial.setSpecularColor(Color.WHITE);
34         sphereMaterial.setSpecularPower(32);
35         sphere.setMaterial(sphereMaterial);
36      }
37   }
```

Fig. 17.18 | Setting the material displayed on 3D shapes.

`PhongMaterial` for the `Box`

Lines 21–23 configure and set the `Box` object's `PhongMaterial`. The **`setDiffuseColor` method** sets the color that's applied to the `Box`'s surfaces (that is, sides). The scene's lighting effects determine the shades of the color applied to each visible surface. These shades change based on the angle from which the light shines on the objects.

`PhongMaterial` for the `Cylinder`

Lines 26–28 configure and set the `Cylinder` object's `PhongMaterial`. The **`setDiffuseMap` method** sets the `Image` applied to the `Cylinder`'s surfaces. Again, the scene's lighting affects how the image is shaded on the surfaces. In the output, notice that the image is darker at the left and right edges (where less light reaches) and barely visible at the bottom (where almost no light reaches).

`PhongMaterial` for the `Sphere`

Lines 31–35 configure and set the `Sphere` object's `PhongMaterial`. We set the diffuse color to red. The **`setSpecularColor` method** sets

the color of a bright spot that makes a 3D shape appear shiny. The `setSpecularPower` **method** determines the intensity of that spot. Try experimenting with different specular powers to observe changes in the bright spot's intensity.

Generative AI

1 Prompt genAIs for a tutorial on using JavaFX camera and lighting effects.

2 Prompt genAIs to discuss JavaFX's 3D transformations.

17.13 FXGL: A Brief Intro to Game Programming with JavaFX

Gaming is one of the most popular and entertaining application-development areas. By 2027, the global gaming market is expected to grow to $363.2 billion dollars,[14] with mobile games accounting for about one-third of the market.[15]

14. "Video Games–Worldwide." Accessed September 30, 2024. https://www.statista.com/outlook/dmo/digital-media/video-games/worldwide.
15. "Mobile Games–Worldwide." Accessed September 30, 2024. https://www.statista.com/outlook/dmo/digital-media/video-games/mobile-games/worldwide.

FXGL Overview

The open-source, cross-platform **FXGL library**[16,17] adds powerful game-development capabilities to JavaFX. FXGL provides a framework for creating 2D games.[18] It combines JavaFX's capabilities with game-development features, including:

16. "FXGL." Accessed September 30, 2024. https://github.com/AlmasB/FXGL.
17. "FXGL 11." Accessed September 30, 2024. https://github.com/AlmasB/FXGL/wiki/FXGL-11.
18. Its 3D features are experimental at the time of this writing.

- a physics engine for replicating real-world physics, including

mass, density, friction, restitution (bounciness), gravity, force, velocity, acceleration and more,

- collision detection,
- sound effects, and
- support for animations and sprites—images or series of images (often represented as one large image) used to add animations that move independently of a game's background.[19]

19. Löw, Andrew. "What is a sprite?" Accessed October 4, 2024. https://www.codeandweb.com/knowledgebase/what-is-a-sprite.

FXGL encapsulates complex game-development tasks—such as rendering, player input handling, game-state management and scene transitions—so you can focus on game logic and design. It also supports advanced features like multiplayer capabilities and particle effects, which can be used to generate complex animations like smoke, fire, rain, snow, explosions and more.[20] FXGL is cross-platform, so the games you create can run on various operating systems, including Windows, macOS and Linux.

20. "Particle System." Wikipedia, Wikimedia Foundation. Accessed October 4, 2024. https://en.wikipedia.org/wiki/Particle_system.

FXGL Resources

There are extensive resources to help you learn FXGL. The **FXGL GitHub repository** provides an overview of the library:

Click here to view code image

```
https://github.com/AlmasB/FXGL
```

and many Java code examples for basic, intermediate and advanced FXGL features:

Click here to view code image

```
https://github.com/AlmasB/FXGL/tree/dev/fxgl-samples/src/main/java
```

The **FXGLGames GitHub Repository** contains source code for many fully implemented games. You can study their code to learn more about FXGL game development and get ideas for building your own games:

Click here to view code image

```
https://github.com/AlmasB/FXGLGames
```

The **FXGL YouTube playlist** has over 100 video tutorials from FXGL's creator— Almas Baimagambetov:

Click here to view code image

```
https://www.youtube.com/
   playlist?list=PL4h6ypqTi3RTiTuAQFKE6xwflnPKyFuPp
```

The **FXGL wiki** contains tutorials and documentation to help you get started in game programming with FXGL:

Click here to view code image

```
https://github.com/AlmasB/FXGL/wiki/FXGL-11
```

17.14 Wrap-Up

In this chapter, we completed our discussion of JavaFX that began in Chapters 15 and 16. Here, we presented various JavaFX graphics and multimedia capabilities. We used external Cascading Style Sheets (CSS) to customize the appearance of JavaFX `Node`s, including `Label`s and objects of various `Shape` subclasses. We displayed two-dimensional shapes, including lines, rectangles, circles, ellipses, arcs, polylines, polygons and custom paths.

We applied a transform to a `Node`, rotating 18 `Polygon` objects around a specific point to create a circle of star shapes. We created a simple video player using class `Media` to specify the video's location, class `MediaPlayer` to load the video and control its playback and class `MediaView` to display the video.

We animated `Node`s with `Transition` and `Timeline` animations that change `Node` properties to new values over time. We used built-in `Transition` animations to change specific JavaFX `Node` properties (such as a `Node`'s stroke and fill colors, opacity, rotation angle and scale). We used `Timeline` animations with `KeyFrame`s to bounce a `Circle` around a window, demonstrating that such animations can be used to change any modifiable `Node` property. We created frame-by-frame animations with `AnimationTimer`, and we applied JavaFX CSS transitions to a `Node`'s CSS properties.

Next, we presented various capabilities for drawing on a `Canvas` `Node` programmatically using a `GraphicsContext` object, which supported many of the same drawing characteristics and shapes as `Shape` `Node`s. Finally, we demonstrated the three-dimensional

shapes `Box`, `Cylinder` and `Sphere`, and used materials to apply color and images to them. For more information on JavaFX, visit

Click here to view code image

```
https://www.jfx-central.com/
https://foojay.io/today/category/javafx/
```

We briefly introduced FXGL for JavaFX game development.

In the next chapter, you'll build API-based Java generative AI applications using popular generative AIs for tasks like transcribing audio files via speech-to-text APIs, summarizing text, extracting key points from text, performing sentiment analysis on text and translating text to other languages via chat completion APIs, synthesizing audio from text via text-to-speech APIs, generating images based on text prompts via image generation APIs, and more.

Special Section: Generative AI Exercises

For each of the following questions, try multiple genAIs to determine which performs best. Always verify the results. Keep in mind that they sometimes make mistakes and even hallucinate, creating unreliable information.

17.1 ***(JavaFX `AudioClip`s)*** Prompt genAIs for an overview of playing audio with JavaFX `AudioClip`s. Use what you learn to build a simple audio player that allows users to play, pause and stop audio files.

17.2 ***(JavaFX Image Effects)*** Prompt genAIs for an overview of JavaFX's effects you can apply to images. Use what you learn to build an app that enables the user to select an image file, then select an effect to apply to it. Allow the user to adjust effect-specific settings.

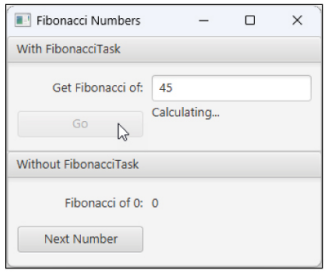
17.3 ***(JavaFX `MediaPlayer` Events)*** Prompt genAIs for an overview of the JavaFX `MediaPlayer`'s events.

17.4 ***(JavaFX `MediaPlayer` Errors)*** Prompt genAIs for an overview of the possible errors that might occur when using a JavaFX `MediaPlayer` and how you should handle them.

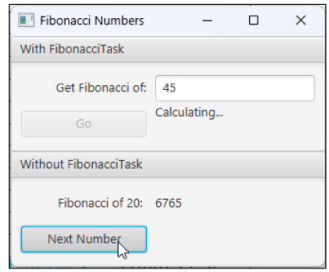
17.5 ***(JavaFX `Node` Transformations)*** Prompt genAIs, with the code from Section 17.5, asking them to enhance the JavaFX application so it demonstrates all of JavaFX's `Node` transformations. Compile the application and try it.

17.6 ***(JavaFX 3D Lighting and Camera Effects)*** Prompt genAIs with the FXML and Java code from Section 17.12, asking them to modify the example to demonstrate JavaFX lighting and camera effects.

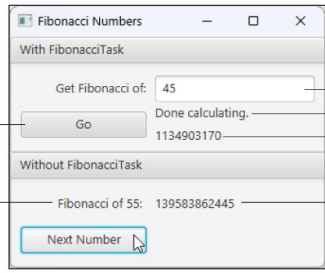
a) Begin calculating Fibonacci of 45 in the background

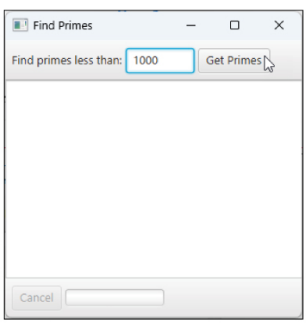


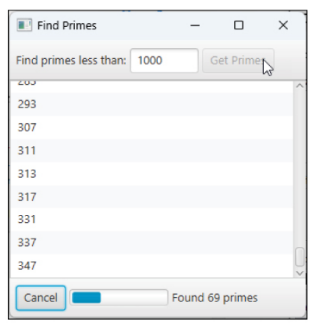
b) Calculating other Fibonacci values while Fibonacci of 45 continues calculating

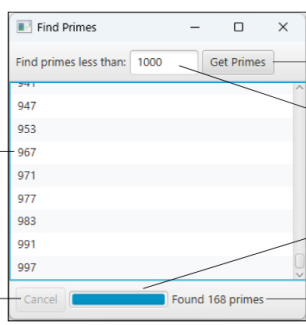


c) Fibonacci of 45 calculation finishes









Concurrency: Platform Threads to Virtual Threads

Objectives

In this chapter, you'll:

- Understand concurrency, parallelism and multithreading.
- Compare the performance of `Arrays` methods `sort` and `parallelSort`.
- Use parallel streams for better multi-core performance.
- Use `ExecutorService` to launch concurrent threads that execute `Runnable`s.
- Be introduced to Project Loom for massively concurrent, high-throughput applications.
- Launch tasks with lightweight virtual threads.
- Group tasks and subtasks via structured concurrency.
- Use structured concurrency and scoped values to share immutable data among tasks.
- Understand producer/consumer relationships.
- Coordinate access to shared mutable data via classes of `java.util.concurrent`.
- Update JavaFX's GUIs in a thread-safe manner.
- Return intermediate results from a long-running task to a JavaFX GUI.

Outline

18.1 Introduction

18.2 `sort/parallelSort` Timings with the Date/Time API

18.3 Sequential vs. Parallel Streams

18.4 Creating and Executing Platform Threads with the `Executor` Framework

18.5 Project Loom Overview

18.6 Creating and Executing Virtual Threads with the `Executor` Framework

18.6.1 Launching Virtual Threads with a Thread-Per-Task Executor

18.6.2 Other Ways to Launch Virtual Threads

18.7 Profiling Platform vs. Virtual Threads

18.7.1 `Future` and `Callable` Interfaces

18.7.2 Threading Approaches We'll Use

18.7.3 Launching and Timing Large Numbers of Concurrent Tasks

18.7.4 Testing the Threading Approaches

18.7.5 Performance Analysis

18.8 Structured Concurrency and Scoped Values

18.9 Thread Synchronization Overview

18.10 Producer/Consumer Relationship with `ArrayBlockingQueue`

18.10.1 Synchronization and State Dependence

18.10.2 Class `ArrayBlockingQueue`

18.10.3 Overview of the Producer/Consumer Example

18.10.4 Interface `Buffer`

18.10.5 Class `BlockingBuffer`

18.10.6 Class `BlockingBufferTest`

18.10.7 Bounded Buffers

18.10.8 Other Concurrent Collections

18.11 Multithreading in JavaFX

18.11.1 Performing Computations in a Worker Thread: Fibonacci Numbers

18.11.2 Processing Intermediate Results: Sieve of Eratosthenes
18.12 Wrap-Up
Special Section: Generative Exercises

18.1 Introduction

It would be nice if we could focus on performing only one task at a time and doing it well. That's usually difficult to do in a complex world with so much happening simultaneously. This chapter presents Java's capabilities for creating and managing multiple tasks. As we'll demonstrate, this can significantly improve program performance and responsiveness.

Concurrency

When we say that two tasks operate **concurrently**, we mean they're both making progress. Until the early 2000s, most computers had only a single processor. Operating systems on such computers executed tasks concurrently by rapidly switching between them, doing a small portion of each before moving on to the next, so all tasks keep progressing. For example, it's common for personal computers to concurrently accept keyboard input, compile a program, send a file to a printer, receive electronic mail messages over a network and more. Since its inception, Java has supported concurrency.

Parallelism

When we say that two tasks operate **in parallel**, we mean they're executing simultaneously. In this sense, parallelism is a subset of concurrency. The human body performs a great variety of operations in parallel. Respiration, blood circulation, digestion, thinking and walking, for example, can occur in parallel, as can all the senses—sight, hearing, touch, smell and taste. It's believed that this parallelism is possible because the human brain is thought to contain billions of "processors." Today's multi-core CPUs have multiple processors that can perform tasks in parallel. A key difference between concurrency and parallelism is that code can execute concurrently by sharing a single core, whereas executing

code in parallel requires multiple cores.

Java Concurrency

Java makes concurrency available to you through the language and APIs—we use only API concurrency features in this chapter. Java programs can have multiple **threads of execution**, each with its own method-call stack, allowing it to execute concurrently with other threads while sharing system resources such as processors, memory and file handles. This is called **multithreading**. The JVM creates threads to run programs and perform housekeeping tasks such as garbage collection.

A problem with single-threaded programs that can lead to poor responsiveness is that lengthy activities must complete before others can begin. In a multithreaded program, threads can be distributed across multiple cores (if available) so multiple tasks execute in parallel. This enables programs to operate more efficiently and optimize system resource use. Multithreading can also increase performance on single-processor systems—when one thread cannot proceed (because, for example, it's waiting for the result of an I/O operation), another can use the processor.

Concurrent Programming Uses

We'll discuss many applications of **concurrent programming**. For example, when streaming an audio or video over the Internet, the user may not want to wait until the entire audio or video downloads before starting the playback. To solve this problem, multiple threads can be used:

- one to download the audio or video (later in the chapter, we'll refer to this as a producer) and
- another to play it (later in the chapter, we'll refer to this as a consumer).

These threads proceed concurrently. To avoid choppy playback, an audio/video player **synchronizes** the threads' actions (that is, coordinates them) so the player thread doesn't begin playing until there's enough audio or video in memory to keep the player thread busy. Producer and consumer threads share memory—we'll demonstrate one of Java's concurrent collections that automatically coordinates such threads to ensure correct execution.

Concurrent Programming Is Difficult

Writing multithreaded programs can be error-prone. Although the human mind can perform functions concurrently, jumping between parallel trains of conscious thought can be difficult.

To understand why multithreaded programs can be challenging to write and comprehend, try the following experiment: Open three books to page 1 and try reading the books concurrently. Read a few words from the first book, then a few from the second, then a few from the third, then loop back and read the next few words from the first book, and so on. After this experiment, you'll appreciate many of the challenges of multithreading— switching between the books, reading briefly, remembering your place in each book, moving the book you're reading closer so you can read it and pushing the books you're not reading aside—and, amid all this chaos, trying to comprehend the content of the books!

The reading-multiple-books analogy is similar to **context switching** in multithreaded applications. To switch between threads, your system must pause one thread, save its state, load another thread's state and resume that thread's execution. This context switching affects performance due to the time spent managing threads instead of running code. Frequent context switching makes it harder to process information efficiently.

Use the Prebuilt Classes of the Concurrency APIs Whenever Possible

In programs requiring thread synchronization, most programmers should use the Java Concurrency APIs from **package** `java.util.concurrent` that manage synchronization for you, like `ArrayBlockingQueue` (Section 18.10). The concurrency API classes are

- written by experts,
- designed to be correct and operate efficiently,
- have been thoroughly tested and debugged, and
- help you avoid common traps and pitfalls.

Section 18.10.8 briefly overviews the Java Concurrency API's other prebuilt concurrent collections.

Project Loom: Virtual Threads, Structured Concurrency and Scoped Values

In addition to several traditional concurrency and multithreading topics, this chapter focuses on Project Loom's enhancements to Java's threading model that provide simpler, lightweight virtual threads (finalized in Java 21). These require significantly less memory than Java's traditional threads (known as platform threads), enabling applications to launch enormous numbers of concurrent virtual threads to meet the demands of today's massively parallel applications. Project Loom also reduces the complexity of writing concurrent programs, making concurrency easier to use and get right.

We also introduce two emerging Project Loom technologies—structured concurrency and scoped values.[1] Structured concurrency enables programmers to treat multiple threads as a single unit of work (a scope), making it easier to manage those threads, handle errors and reclaim resources no longer needed. Scoped values simplify data sharing among virtual threads in the same scope, reducing the need for complex thread-synchronization techniques. Scoped values become implicit arguments to every method called in a given structured concurrency scope.

1. At the time of this writing, structured concurrency and scoped values were preview features. Once they are finalized, we'll post the updated examples on our blog at `https://deitel.com/blog`.

Generative AI

1 Prompt genAIs to provide real-world analogies for concurrency vs. parallelism.

2 Prompt genAIs for an overview of the problems that might occur if multiple threads share data without synchronization.

18.2 `sort/parallelSort` Timings with the Date/Time API

Let's begin with an example in which Java automatically divides tasks into subtasks that can execute in parallel on multi-core systems. Section 6.19 used class `Arrays`'s `static` method `sort` to sort an array. It also mentioned `static` method `parallelSort` for sorting large arrays more efficiently on multi-core systems. Figure

18.1 compares the performance of `sort` and `parallelSort` by sorting identical 100-million-element arrays of random `int` values, demonstrating `parallelSort`'s performance improvement over `sort`. We ran this example on a MacBook Pro with a 12-core M2 Max processor. We split the code into parts for discussion purposes. Line 11 uses the **`InstantSource` interface's `static` `system` method** to get an `InstantSource` representing the system clock we'll use to time the sorts.

Click here to view code image

```
1  // Fig. 18.1: SortComparison.java
2  // Comparing the performance of Arrays methods sort and parallelSort.
3  import java.time.Duration;
4  import java.time.InstantSource;
5  import java.text.NumberFormat;
6  import java.util.Arrays;
7  import java.util.concurrent.ThreadLocalRandom;
8
9  public class SortComparison {
10    public static void main(String[] args) {
11       var clock = InstantSource.system(); // system clock for timing sorts
12
```

Fig. 18.1 | Comparing the performance of `Arrays` methods `sort` and `parallelSort`.

Creating and Filling the Arrays

Line 16 creates a 100,000,000-element array of `int`s. Lines 17–18 fill the array with random `int` values using **`Arrays` `static` method `parallelSetAll`**, which splits the work of filling the array with values across multiple threads that can operate in parallel on your computer's CPU cores. This makes filling the array faster for large arrays. The method receives as arguments the array to fill and a **generator function** that produces a value used to fill an array element. We're filling an `int` array, so generator must be an `IntUnaryOperator` that receives an `int` and returns an `int`. Depending on which `parallelSetAll` overload you use, the generator function must implement one of the following `java.util.function` interfaces

- `IntUnaryOperator` for `int` arrays,
- `IntToLongFunction` for `long` arrays,
- `IntToDoubleFunction` for `double` arrays or

- `IntFunction<T>` for arrays of any non-primitive type.

In this example, we implemented `IntUnaryOperator` (introduced in Section 14.9) as a lambda expression (line 18) that calls

Click here to view code image

```
ThreadLocalRandom.current().nextInt()
```

to choose a random `int` value for each array element.[2] At the end of this section, we'll explain why we used `ThreadLocalRandom` rather than `RandomGenerator`. Line 19 calls `array1`'s `clone` method to make a copy so the calls to `sort` and `parallelSort` sort the same values.

2. The lambda's parameter, `i`, is not used in this program. For such cases, you can replace the variable name with an underscore (_).

Click here to view code image

```
13          // create array of random ints, then copy it; Arrays method
14          // parallelSetAll divides the task of randomly initializing
15          // array1 across available cores
16          var array1 = new int[100_000_000];
17          Arrays.parallelSetAll(array1,
18             i -> ThreadLocalRandom.current().nextInt());
19          var array2 = array1.clone();
20
```

Timing `Arrays` Method `sort`

Lines 23 and 25 use the `clock` object's **`instant` method** to get the current system time before and after the `sort` call. This method returns an **`Instant`** representing the current time. Line 28 determines the difference between two `Instant`s by calling **`Duration`'s `static` method `between`**, which returns a `Duration` object. **`Duration` method `toMillis`** converts the time difference to milliseconds. Finally, line 29 calculates and displays the sort time in seconds. In this sample execution, `sort` took 9.1 seconds to sort the array.

Click here to view code image

```
21          // time the sorting of array1 with Arrays method sort
22          System.out.println("Starting sort");
23          var sortStart = clock.instant();
24          Arrays.sort(array1);
25          var sortEnd = clock.instant();
```

```
26
27          // display timing results
28          long sortTime = Duration.between(sortStart, sortEnd).toMillis();
29          System.out.printf("Sort time: %.3f seconds%n%n", sortTime / 1000.0);
30
```

```
Starting sort
Sort time: 9.100 seconds
```

Timing `Arrays` Method `parallelSort`

Lines 32–41 use the same techniques as lines 22–29 to time the call to `Arrays` method `parallelSort` and display the sort time in seconds. **Method `parallelSort` took less than a second to sort the same array—a significant performance improvement.**

Click here to view code image

```
31          // time the sorting of array2 with Arrays method parallelSort
32          System.out.println("Starting parallelSort");
33          var parallelSortStart = clock.instant();
34          Arrays.parallelSort(array2);
35          var parallelSortEnd = clock.instant();
36
37          // display timing results
38          long parallelSortTime =
39             Duration.between(parallelSortStart, parallelSortEnd).toMillis();
40          System.out.printf("Sort time: %.3f seconds%n%n",
41             parallelSortTime / 1000.0);
42
```

```
Starting parallelSort
Sort time: 0.965 seconds
```

Displaying the Percentage Difference Between the Sort Times

Lines 44–45 use `NumberFormat` (from package `java.text`) to format the ratio of the sort times as a percentage. `NumberFormat` `static` method `getPercentInstance` returns a `NumberFormat` for formatting percentages. The object's `format` method performs the formatting. Lines 46–47 display the formatted percentage. In this sample execution, `sort` took 843% more time than `parallelSort`.[3]

3. Depending on your computer's capabilities, you might observe slower or faster performance than what we observed on our test computer.

Click here to view code image

```
43          // display time difference as a percentage
44          String percentage = NumberFormat.getPercentInstance().format(
45             (double) (sortTime - parallelSortTime) / parallelSortTime);
46          System.out.printf("sort took %s more time than parallelSort%n",
47             percentage);
48       }
49    }
```

```
sort took 843% more time than parallelSort
```

ThreadLocalRandom VS. RandomGenerator

Java's regular random-number generators, like the one returned by

```
RandomGenerator.getDefault()
```

are not designed to be shared among multiple threads. They are not "**thread-safe**"—when multiple threads share a random-number generator, they can interfere with one another, making the generated values less random. Each thread that calls **`ThreadLocalRandom`'s `static` method `current`** receives its own `RandomGenerator` object. This prevents conflicts when `parallelSetAll` generates random numbers for multiple portions of an array in parallel.

Generative AI

1 Prompt genAIs for guidelines on when to use `parallelSort` vs. `sort` for better performance.

2 Prompt genAIs for guidelines on when to use `parallelSetAll` vs. a loop to initialize an array's elements.

3 Prompt genAIs, asking how multiple threads sharing a regular Java `RandomGenerator` object can interfere with one another.

18.3 Sequential vs. Parallel Streams

Let's continue with another example in which Java handles a task's parallelization for you. Chapter 14 presented Java's functional programming techniques using lambdas and streams. We mentioned that streams are easy to parallelize, enabling better performance on multi-core systems. Figure 18.2 uses the timing capabilities from Section 18.2 to observe the difference in performance between sequential and parallel stream operations performed on a one-billion-element array of random `long` values. **Depending on your computer's capabilities, you may need to reduce the number of elements in this example—the previous edition of this book used 50 million elements.** To give you a sense of our current test computer's increase in computing power compared to the one we used in our last edition, this example's parallel stream operation took 143 milliseconds for 50 million elements in 2017 and 359 milliseconds for one billion elements (20 times more) in 2025. We split the example into parts for discussion purposes. Lines 14–16 create and fill the array with random `long` values in the range 1–1000. We use type `long` in this example so we can represent the sum of one billion elements in this range of values.

Click here to view code image

```
1  // Fig. 18.2: StreamStatisticsComparison.java
2  // Comparing performance of sequential and parallel stream operations.
3  import java.time.Duration;
4  import java.time.InstantSource;
5  import java.util.Arrays;
6  import java.util.LongSummaryStatistics;
7  import java.util.concurrent.ThreadLocalRandom;
8
9  public class StreamStatisticsComparison {
10    public static void main(String[] args) {
11       var clock = InstantSource.system(); // system clock for timing
12
13       // create array of random long values
14       var values = new long[1_000_000_000];
15       Arrays.parallelSetAll(values,
16          i -> ThreadLocalRandom.current().nextLong(1, 1001));
17
```

Fig. 18.2 | Comparing performance of sequential and parallel stream operations.

Performing Stream Operations with Separate Passes of a Sequential Stream

Section 14.7 demonstrated various numerical operations on `IntStream`s. Lines 19–25 perform and time the `count`, `sum`, `min`, `max` and `average` stream operations, performing each individually on the `LongStream` returned by `Arrays` method `stream`. Lines 28–35 display the results and the total time required to perform all five operations.

Click here to view code image

```
18          // perform calculations separately
19          var separateStart = clock.instant();
20          long count = Arrays.stream(values).count();
21          long sum = Arrays.stream(values).sum();
22          long min = Arrays.stream(values).min().getAsLong();
23          long max = Arrays.stream(values).max().getAsLong();
24          double average = Arrays.stream(values).average().getAsDouble();
25          var separateEnd = clock.instant();
26
27          // display results
28          System.out.println("Calculations performed separately");
29          System.out.printf("    count: %,d%n", count);
30          System.out.printf("      sum: %,d%n", sum);
31          System.out.printf("      min: %,d%n", min);
32          System.out.printf("      max: %,d%n", max);
33          System.out.printf("  average: %f%n", average);
34          System.out.printf("Total time in milliseconds: %d%n%n",
35          Duration.between(separateStart, separateEnd).toMillis());
36
```

```
Calculations performed separately
    count: 1,000,000,000
      sum: 500,490,878,099
      min: 1
      max: 1,000
  average: 500.490878
Total time in milliseconds: 4052
```

Performing Stream Operations with a Single Pass of a Sequential Stream

Streams are processed sequentially by default. Lines 39–47 demonstrate the performance improvement you get by using `LongStream` method `summaryStatistics`, which determines the count, sum, minimum value, maximum value and average in one pass of a `Long-Stream`. This operation took approximately 85% of the time required to perform the five operations separately.

Click here to view code image

```
37          // time summaryStatistics operation with sequential stream
38          System.out.println("Calculating statistics on sequential stream");
39          var sequentialStart = clock.instant();
40          LongSummaryStatistics results1 =
41             Arrays.stream(values).summaryStatistics();
42          var sequentialEnd = clock.instant();
43
44          // display results
45          displayStatistics(results1);
46          System.out.printf("Total time in milliseconds: %d%n%n",
47             Duration.between(sequentialStart, sequentialEnd).toMillis());
48
```

```
Calculating statistics on sequential stream
Statistics
    count: 1,000,000,000
      sum: 500,490,878,099
      min: 1
      max: 1,000
  average: 500.490878
Total time in milliseconds: 3451
```

Performing Stream Operations with a Single Pass of a Parallel Stream

Lines 51–58 demonstrate the performance improvement from calling `summaryStatistics` on a parallel stream that automatically takes advantage of multi-core processors. To obtain a parallel stream, invoke the **`parallel` method** on a stream object. The sample output indicates that performing the operations on a parallel stream decreased the total calculation time even further. Calling `summaryStatistics` on a parallel stream took 10.4% of the time it took on the sequential stream and just 8.9% of the time it took to perform the individual operations sequentially.

Click here to view code image

```
49          // time sum operation with parallel stream
50          System.out.println("Calculating statistics on parallel stream");
51          var parallelStart = clock.instant();
52          LongSummaryStatistics results2 =
53             Arrays.stream(values).parallel().summaryStatistics();
54          var parallelEnd = clock.instant();
55
```

```
56          // display results
57          displayStatistics(results2);
58          System.out.printf("Total time in milliseconds: %d%n%n",
59             Duration.between(parallelStart, parallelEnd).toMillis());
60       }
61
```

```
Calculating statistics on parallel stream
Statistics
    count: 1,000,000,000
      sum: 500,490,878,099
      min: 1
      max: 1,000
  average: 500.490878
Total time in milliseconds: 359
```

Method `displayStatistics`

The `displayStatistics` method displays the count, sum, min, max and average stored in its `LongSummaryStatistics` argument.

Click here to view code image

```
62       // display's LongSummaryStatistics values
63       private static void displayStatistics(LongSummaryStatistics stats) {
64          System.out.println("Statistics");
65          System.out.printf("    count: %,d%n", stats.getCount());
66          System.out.printf("      sum: %,d%n", stats.getSum());
67          System.out.printf("      min: %,d%n", stats.getMin());
68          System.out.printf("      max: %,d%n", stats.getMax());
69          System.out.printf("  average: %f%n", stats.getAverage());
70       }
71    }
```

Generative AI

1 Prompt genAIs to discuss the pros and cons of parallelism in Java functional programming with lambdas and streams.

2 Prompt genAIs for guidelines on when to use sequential vs. parallel streams in Java functional programming with lambdas and streams.

3 Prompt genAIs asking about parallel stream performance considerations for working with various Java collections.

18.4 Creating and Executing Platform Threads with the `Executor` Framework

In Sections 18.2–18.3, we used parallel methods of the `Arrays` class and parallel streams that automatically "figure out" how to take advantage of your system's CPU cores to improve performance. You can also create your own custom concurrent tasks.

Creating Concurrent Tasks with the `Runnable` Interface

You implement the **`Runnable` interface** (of package `java.lang`) to specify a task that can execute concurrently with other tasks. You specify the code that performs your task by overriding the **`Runnable` interface's `run` method**.

Executing `Runnable` Objects with an `Executor`

You execute a `Runnable` to perform its task, typically using an object that implements the **`Executor` interface**. This interface's **`execute` method** receives a `Runnable` and executes it at some point in the future. When an `Executor` begins executing a `Runnable`, it assigns the `Runnable` to a `Thread` and calls the `Runnable`'s `run` method to perform the specified task. Any method called from a `Runnable`'s `run` method executes as part of that task's `Thread`.

You obtain an `Executor` object by calling a `static` method of the **`Executors` class** (package `java.util.concurrent`). This class's methods return objects that implement the **`ExecutorService` interface** (package `java.util.concurrent`), which extends `Executor` and declares additional methods for managing an `Executor`'s life cycle. `ExecutorService` objects are `AutoCloseable`, so you can use them with the `try`-with-resources statement.

Though it's possible to create `Thread`s explicitly, `Executor`s have many advantages. For example, some `Executor`s manage a group of platform threads called a **thread pool** and can reuse existing platform threads to eliminate the overhead of creating new ones for each task. Such `Executor`s can improve performance for CPU-intensive tasks by optimizing the number of threads to ensure that the CPU's cores stay busy.

Implementing the `Runnable` Interface

Let's use a mechanical example to demonstrate creating and executing `Runnable` tasks. Class `PrintTask` (Fig. 18.3) implements `Runnable` (line 5)—each `PrintTask` can execute concurrently with other tasks. Variable `sleepTime` (line 8) stores a random integer value from 0 to 5000 milliseconds (line 15)—we did not use `ThreadLocalRandom` here, as the PrintTask constructor is always called from the `main` thread. Each thread running a `Print-Task` sleeps for the amount of time specified by `sleepTime`, then outputs its task's name and a message indicating that it's done sleeping.

Click here to view code image

```
1  // Fig. 18.3: PrintTask.java
2  // PrintTask class sleeps for a random time from 0 to 5 seconds
3  import java.util.random.RandomGenerator;
4
5  public class PrintTask implements Runnable {
6     private static final RandomGenerator generator =
7        RandomGenerator.getDefault();
8     private final int sleepTime; // random sleep time for thread
9     private final String taskName;
1011     // constructor gives the PrintTask a name and
12    // picks random sleep time between 0 and 5 seconds
13    public PrintTask(String taskName) {
14       this.taskName = taskName;
15       sleepTime = generator.nextInt(5001); // milliseconds
16    }
17
18    // method run contains the code that a thread will execute
19    @Override
20    public void run() {
21       try { // put thread to sleep for sleepTime amount of time
22          System.out.printf("%s going to sleep for %d milliseconds.%n",
23             taskName, sleepTime);
24          Thread.sleep(sleepTime); // put thread to sleep
25       }
26       catch (InterruptedException exception) {
27          exception.printStackTrace();
28          Thread.currentThread().interrupt(); // re-interrupt the thread
29       }
30
31       // print task name
32       System.out.printf("%s done sleeping%n", taskName);
33    }
34 }
```

Fig. 18.3 | `PrintTask` class sleeps for a random time from 0 to 5 seconds.

A `PrintTask` executes when a thread calls its `run` method (lines 19–33). Lines 22–23 output a message indicating the task's name and that it's going to sleep for `sleepTime` milliseconds. Line 24 invokes **`Thread` `static` method `sleep`** to pause the thread for the specified amount of time. At this point, the thread loses the processor, and the operating system can allow another thread to execute. When the thread awakens, and the operating system eventually assigns it to a processor to continue executing, line 32 outputs a message indicating that the task is done sleeping, then method `run` terminates.

A `Runnable`'s `run` Method Must Catch `InterruptedException`s

The `catch` handler at lines 26–29 is required because method `sleep` might throw a checked **`InterruptedException`** if a sleeping thread's **`interrupt` method** is called. Recall from Chapter 10 that when you override a method, the `throws` clause may contain only the same exception types or a subset of those declared in the original method's `throws` clause. In the `Runnable` interface's definition, method `run` does not have a `throws` clause, so we cannot add one in line 20. For this reason, you must catch `InterruptedException`s (and any other checked exceptions) in a `Runnable`'s `run` method.

When a thread is interrupted, it's typically because another part of the program needs it to terminate—perhaps due to an error or another thread having already finished the required task. Every thread has a `boolean` interrupted status. A thread can periodically check this status by calling **`Thread` `static` method `isInterrupted`**. If this method returns `true`, the thread knows it has been interrupted and can decide how to terminate gracefully. When you catch an `InterruptedException`, the thread's interrupted status is set to `false`, which could prevent the thread from terminating. For this reason, it's considered good practice to set the thread's interrupt status back to `true` in the `catch` handler to ensure that subsequent calls to `isInterrupted` return `true`. Line 28 obtains a reference to the currently executing `Thread` by calling **`Thread` `static` method `currentThread`**, then uses that `Thread`'s `interrupt` method to set the thread's interrupted status to `true`.[4]

4. For detailed information on handling thread interruptions, read Chapter 7 of *Java Concurrency in Practice* by Brian Goetz, et al., Addison-Wesley Professional, 2006.

Executing `PrintTask`s Concurrently

Figure 18.4 uses an `ExecutorService` object to execute `PrintTask`s. Lines 8–10 create three `PrintTask`s to execute. Line 18 in the `try`-with-resources statement calls **`Executors` `static` method `newCachedThreadPool`** to obtain an `ExecutorService` that manages a cached thread pool. Such a thread pool executes each task by

- using an existing platform `Thread` if one is available in the pool or
- creating a new platform `Thread` if no existing ones are available to reuse.

When a task completes, the cached thread pool `ExecutorService` keeps the platform `Thread` for 60 seconds. If the program submits another task to the `ExecutorService` within that time, it reuses the cached `Thread` to execute that task, eliminating the overhead of creating a new platform thread (we say more about this overhead in Section 18.7). Otherwise, the `ExecutorService` terminates the `Thread` and removes it from the cache. Cached thread pools are typically used in programs that execute many short tasks.[5]

5. "Class `Executors`." Accessed February 7, 2025. https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/util/concurrent/Executors.html#newCachedThreadPool().

Click here to view code image

```
1  // Fig. 18.4: TaskExecutor.java
2  // Using an ExecutorService to execute Runnables.
3  import java.util.concurrent.Executors;
4
5  public class TaskExecutor {
6     public static void main(String[] args) {
7        // create and name each runnable
8        var task1 = new PrintTask("task1");
9        var task2 = new PrintTask("task2");
10       var task3 = new PrintTask("task3");
11
12       System.out.println("Starting Executor\n");
13
14       // create ExecutorService for executing tasks; when the
15       // try-with-resources statement completes, the ExecutorService's
16       // close method shuts it down, preventing it from accepting new
17       // tasks and waiting for existing ones to complete execution
18       try (var executorService = Executors.newCachedThreadPool()) {
19          // start the three PrintTasks
20          executorService.execute(task1); // start task1
```

```
21            executorService.execute(task2); // start task2
22            executorService.execute(task3); // start task3
23         }
24
25         System.out.println("\nMain ends.");
26      }
27   }
```

```
Starting Executor

task1 going to sleep for 4610 milliseconds.
task3 going to sleep for 4833 milliseconds.
task2 going to sleep for 992 milliseconds.
task2 done sleeping
task1 done sleeping
task3 done sleeping

Main ends.
```

```
Starting Executor

task3 going to sleep for 357 milliseconds.
task1 going to sleep for 4670 milliseconds.
task2 going to sleep for 3360 milliseconds.
task3 done sleeping
task2 done sleeping
task1 done sleeping

Main ends.
```

Fig. 18.4 | Using an `ExecutorService` to execute `Runnable`s.

Java maps each `Thread` in a cached thread pool to an operating system thread—the Java documentation refers to these as **platform threads**. Multi-core systems can schedule multiple platform threads (and the other programs running on your computer) on separate cores, enabling them to execute in parallel. Lines 20–22 use the cached thread pool to execute `Runnable`s—in this case, `PrintTask`s—at some time in the future. Method `execute` returns immediately from each invocation—the program does not wait for each `PrintTask` to finish.

Waiting for Tasks to Complete Execution

After scheduling tasks to execute, you'll typically want to wait for them to complete—for example, so you can use the tasks' results, as

we'll do in later examples. When the `try`-with-resources statement completes execution (line 23), it calls the **`ExecutorService`'s `close` method**, which

- calls **`ExecutorService` method `shutdown`** to prevent the `ExecutorService` from accepting new tasks then
- calls **`ExecutorService` method `awaitTermination`** to wait for all the previously scheduled tasks to complete.

Main Thread

The code in `main` executes in the **`main` thread**, which is created by the JVM. The code in `PrintTask`'s `run` method (lines 19–33 of Fig. 18.3) executes sometime after each task is passed to the `ExecutorService`'s `execute` method (Fig. 18.4, lines 20–22).

Sample Outputs

Each task displays its name and sleep time then puts its thread to sleep. In most cases, the thread running the task with the shortest sleep time awakens first (though this is not guaranteed), then indicates it's done sleeping and terminates. In sample outputs, note that the tasks did not go to sleep in the same order they were launched. This illustrates that **we cannot predict the order in which the tasks will start executing, even if we know the order in which they were created and started.** Similarly, **we cannot predict when a thread that awakens from a sleep will be assigned to a processor to continue executing.**

Generative AI

1 Prompt genAIs for `InterruptedException` use-cases and a code example demonstrating how a `Runnable` can terminate gracefully when an `InterruptedException` occurs.

2 Prompt genAIs, asking them to list the thread pool types class `Executors` provides and why you'd use each.

3 Prompt genAIs for `ExecutorService` best practices.

4 Prompt genAIs, asking why, when you put several threads with different sleep times to sleep, it's not guaranteed that the one with the shortest sleep time will awaken first.

18.5 Project Loom Overview

Section 18.4 executed `PrintTask`s using resource-intensive platform threads. Each platform thread has its own default method-call stack size—typically one to two megabytes (depending on the hardware)[6] to keep track of methods called by the task in that thread. The number of platform threads is limited not only by the amount of available memory but also by[7]

6. "The `java` Command." Accessed February 9, 2025. https://docs.oracle.com/en/java/javase/23/docs/specs/man/java.html.
7. Summary of responses from *ChatGPT*, *Claude*, *Gemini* and *Perplexity* to the prompt "What are the operating system limits for threads in an application?" https://chatgpt.com, https://claude.ai, https://gemini.google.com/app and https://perplexity.ai.

   - operating system limits on threads per process (i.e., per application) and
   - the overheads of scheduling and executing large numbers of concurrent tasks and sharing resources among those tasks.

These limitations make it difficult for developers to build scalable applications requiring thousands or even millions of concurrent tasks—like those that handle the enormous number of interactions between users and popular social media apps, such as YouTube, Face-book, WhatsApp, Instagram, Snapchat, TikTok, Reddit and X. In Java, such applications have typically required complex programming techniques and frameworks to reduce overhead and share limited resources.

Project Loom enhances Java with three new concurrency capabilities that simplify massively parallel application development:

- virtual threads (Sections 18.6–18.7),[8]
- structured concurrency (Section 18.8)[9] and
- scoped values (Section 18.8).[10]

8. "JEP 444: Virtual Threads." Accessed February 9, 2025. https://openjdk.org/jeps/444.
9. "JEP 499: Structured Concurrency (Fourth Preview)." Accessed

February 9, 2025. https://openjdk.org/jeps/499.

10. "JEP 487: Scoped Values (Fourth Preview)." Accessed February 9, 2025. https://openjdk.org/jeps/487.

As we'll discuss in Section 18.6.1, **virtual threads do not replace platform threads—they have different use-cases.**

Virtual threads were finalized in Java 21. Structured concurrency and scoped values were preview features at the time of this writing. If our code examples require changes when the final structured concurrency and scoped values versions are released, we'll post updates on our blog at https://deitel.com/blog.

Virtual Threads Have Significantly Less Overhead than Platform Threads

Virtual threads are lightweight and managed by the JVM, not the operating system. Rather than pre-allocating a large method-call stack, the JVM allocates a few-hundred-byte **stack chunk** for each virtual thread. This stack chunk grows and shrinks dynamically based on the virtual thread's required number of method-call stack frames—possibly as few as one if the task performs all its work in one method's body.[11,12] Due to the low virtual-thread memory requirements, an application can create thousands or even millions of virtual threads without overwhelming system resources. Creating millions of platform threads would quickly exhaust system resources, possibly causing the program to crash with an `OutOfMemoryError`.

11. "JEP 444: Virtual Threads." Accessed February 9, 2025. https://openjdk.org/jeps/444.

12. Goetz, Brian. "Virtual Threads: New Foundations for High-Scale Java Applications." InfoQ, November 2, 2022. Accessed February 9, 2025. https://www.infoq.com/articles/java-virtual-threads/.

Generative AI

1 Prompt genAIs, asking them to explain any limitations on the number of platform threads for the latest versions of macOS and Windows.

18.6 Creating and Executing Virtual Threads with the `Executor` Framework

Java provides several ways to execute tasks with virtual threads. Section 18.6.1 modifies Section 18.4's `PrintTask` example to demonstrate an `ExecutorService` that creates a separate virtual thread for each task. Section 18.6.2 shows how to create `Thread` objects for virtual threads.

18.6.1 Launching Virtual Threads with a Thread-Per-Task Executor

A primary virtual threads use-case is for server software that handles massive numbers of concurrent requests where each spends most of its time blocked from continuing to execute until I/O requests complete, such as

- web servers that respond to user requests by returning web pages to their web browsers or
- application servers that return the results of database queries to applications.

Platform threads are more appropriate for CPU-intensive tasks, such as sorting and long-running calculations. `Arrays` method `parallelSort` uses platform threads to execute the subtasks that sort portions of an array.

As an example of massive numbers of concurrent requests, consider the social media site X.com. In December 2024, X had 4.6 billion visits, and each visitor viewed 12.35 pages on average.[13] If we assume each page view requires one server request,[14] there were approximately 56.81 billion requests (4.6 billion times 12.35) handled by `X.com`'s servers—about 21,200 requests per second. Lightweight virtual threads enable Java servers to handle huge numbers of concurrent requests like this with **thread-per-request code**—that is, creating a new virtual thread for each request.

13. Similarweb, "X.com Traffic Analytics & Market Share." Accessed February 9, 2025. `https://www.similarweb.com/website/x.com/#overview`.
14. A request for web-based content often requires many sub-requests for items like images, videos and advertisements that a web browser renders in a web page.

Launching Virtual Threads with an `ExecutorService`

Figure 18.5 reuses Fig. 18.3's `PrintTask` class and launches virtual

threads. Line 18 of Fig. 18.5 calls the **`Executors` class's `static` `newVirtualThreadPerTaskExecutor` method** to get an `ExecutorService` that launches a new virtual thread for each task you submit. Lines 20–22 call the `ExecutorService`'s `execute` method to launch each `PrintTask`.

Click here to view code image

```
1  // Fig. 18.5: VirtualThreadDemo.java
2  // Using virtual threads to execute Runnables.
3  import java.util.concurrent.Executors;
4
5  public class VirtualThreadDemo {
6     public static void main(String[] args) {
7        // PrintTask constructor assigns random sleep time to each
8        PrintTask task1 = new PrintTask("task1");
9        PrintTask task2 = new PrintTask("task2");
10       PrintTask task3 = new PrintTask("task3");
11
12       System.out.println("Starting Virtual-Thread-Per-Task Executor\n");
13
14       // The ExecutorService returned by newVirtualThreadPerTaskExecutor
15       // runs each task in a virtual thread. The try-with-resources block
16       // calls ExecutorService's close method when the block terminates.
17       // The program then waits for the virtual threads to complete.
18       try (var executor = Executors.newVirtualThreadPerTaskExecutor()) {
19          // start the three PrintTasks
20          executor.execute(task1); // start task1
21          executor.execute(task2); // start task2
22          executor.execute(task3); // start task3
23       }
24
25       System.out.println("\nMain ends.");
26    }
27 }
```

```
Starting Virtual-Thread-Per-Task Executor

task2 going to sleep for 2138 milliseconds.
task3 going to sleep for 4525 milliseconds.
task1 going to sleep for 90 milliseconds.
task1 done sleeping
task2 done sleeping
task3 done sleeping

Main ends.
```

```
Starting Virtual-Thread-Per-Task Executor

task1 going to sleep for 3128 milliseconds.
task2 going to sleep for 3661 milliseconds.
task3 going to sleep for 1080 milliseconds.
task3 done sleeping
task1 done sleeping
task2 done sleeping

Main ends.
```

Fig. 18.5 | Using virtual threads to execute `Runnable`s.

Carriers, Mounting and Unmounting

A virtual thread's **carrier** is the platform thread that runs the virtual thread. When the JVM schedules a virtual thread for execution, it **mounts** the virtual thread on a carrier. Then, the operating system is responsible for scheduling the carrier's execution. The JVM **unmounts** a virtual thread from its carrier when it blocks for I/O, sleeps or completes its task. The runtime can then mount a different virtual thread on that carrier to efficiently use the system's platform-thread resources.

Non-Daemon vs. Daemon Threads[15]

15. "Class `Thread`." Accessed February 9, 2025. https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/lang/Thread.html.

Platform threads may be **non-daemon** or **daemon threads**. A Java app runs while its non-daemon threads are still executing their tasks. Examples of non-daemon threads include:

- the thread where the `main` method executes and
- a JavaFX app's event-handling thread.

A daemon thread typically performs background tasks that support a program's execution. Examples of daemon threads include:

- Java's background garbage collector thread that reclaims unused objects from memory and
- an "auto-save" background thread that watches for changes you make to a document and automatically saves them—a common feature in many apps that process files like documents, spreadsheets and more.

A Java app terminates if the only remaining threads are daemon threads.

Thread Defaults

By default, a platform thread is a non-daemon thread unless you

- launch it from a daemon thread or
- call `setDaemon(true)` on it to make it a daemon thread.

Every virtual thread is always a daemon thread.

Awaiting Virtual Thread Completion

If you want your program to remain running until its virtual threads' tasks complete, you must prevent `main` from terminating. The `try`-with-resources statement and `ExecutorService` manage this for you. When the `try`-with-resources statement terminates, it automatically calls the `ExecutorService`'s `close` method, which, in turn, calls

- the `ExecutorService`'s `shutdown` method to prevent the `ExecutorService` from accepting new tasks and
- the `ExecutorService`'s `awaitTermination` method to wait for the `ExecutorSer-vice`'s previously launched tasks to complete execution.

Generative AI

1 Prompt genAIs, asking for examples of non-daemon threads in Java applications.

2 Prompt genAIs, asking why Java virtual threads are always daemon threads.

3 Prompt genAIs, asking why you cannot use `Thread` method `setDaemon` to make a virtual thread a non-daemon thread.

18.6.2 Other Ways to Launch Virtual Threads

`Thread` Method `startVirtualThread`

Platform threads and virtual threads are each instances of the

`java.lang` package's `Thread` class, but a virtual thread is managed entirely by the JVM. If you need

- more control over a virtual thread via class `Thread`'s methods or
- a convenient way to launch a virtual thread without using an `ExecutorService`,

you can use **class `Thread`'s `static` method `startVirtualThread`**. This method receives a `Runnable`, launches a virtual thread to execute it and returns a reference to the virtual thread's `Thread` object. For example, the following statement creates a virtual thread for the `PrintTask` object `task1`:

Click here to view code image

```
var virtualThread = Thread.startVirtualThread(task1);
```

In this case, if your program needs the task's result, you must somehow wait for it to complete—typically by calling the `Thread`'s `join` method before `main` terminates, as in:

```
virtualThread.join();
```

Otherwise, `main` might terminate before the virtual thread executes its task.

Thread Method `ofVirtual`

When you need to specify virtual threads' configuration options, you can use **class `Thread`'s `static` method `ofVirtual`**, which returns a **`Thread.Builder.OfVirtual` object**.[16,17] This object uses the **builder design pattern**—a common technique for constructing complex objects step-by-step via chained method calls.[18,19] The builder design pattern is handy for flexible object configuration—you call only the builder's methods that add the options you need. Without this pattern, a Java class might require many overloaded constructors for each possible combination of options.

16. "Interface `Thread.Builder.OfVirtual`." Accessed February 9, 2025. https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/lang/Thread.Builder.OfVirtual.html.
17. `Thread`'s `static` method `ofPlatform` returns a `Thread.Builder.OfPlatform` object for similar creation and configuration of platform threads.
18. Bloch, Joshua. *Effective Java*. 3rd ed. Upper Saddle River, NJ: Addison-

Wesley, 2018.

19. Kiwy, Frank. “Exploring Joshua Bloch’s Builder design pattern in Java.” May 28, 2021. Accessed February 9, 2025. https://blogs.oracle.com/javamagazine/post/exploring-joshua-blochs-builder-design-pattern-in-java.

A `Thread.Builder.OfVirtual` object enables you to specify several virtual thread options, including:

- the virtual thread’s name,
- a virtual thread’s name concatenated with a specified counter value—useful when launching multiple virtual threads to give each a unique name and
- a method to call when an uncaught exception occurs in a virtual thread.

In the builder design pattern, you call a `static` method to get a builder object. Each subsequent method you call on the builder object adds a configuration option and returns a reference to the builder object so you can add more options. In the case of `Thread.Builder.OfVirtual` object, the last call in the chain is one of the following:

- **`start`** creates a virtual thread to execute its `Runnable` argument, schedules the thread for execution and returns a reference to the corresponding `Thread` object.
- **`factory`** returns a reference to a `ThreadFactory` that can create virtual threads with the settings specified in the previous calls to the builder object’s methods.
- **`unstarted`** creates a virtual thread to execute its `Runnable` argument and returns a reference to the `Thread` object but does not start it. In this case, you must subsequently call the `Thread`’s `start` method to execute the virtual thread.

For example, assuming the variable `id` contains `1` and `task` is a `Runnable` object, the following chained method call would create a new virtual thread named `"Task1"`, schedule the `Runnable` for execution and return a reference to the corresponding `Thread` object:

```
Thread thread =
   Thread.ofVirtual()
         .name("Task", id)
         .start(task);
```

Again, you must call `thread.join()` if you want to ensure the thread executes before the program terminates.

Generative AI

1 Prompt genAIs, asking why you typically call a `static` method to get a builder object.

2 Prompt genAIs, asking for an explanation of the Java method naming convention in which method names begin with the word `"of"`.

3 Prompt genAIs, asking for a few examples of the builder design pattern in the Java APIs.

4 Prompt genAIs for the pros and cons of the three ways to create Java virtual threads.

18.7 Profiling Platform vs. Virtual Threads[20]

20. This section's example was inspired by Brian Goetz's intro to virtual threads in the article "Virtual Threads: New Foundations for High-Scale Java Applications." InfoQ, November 2, 2022. Accessed January 31, 2025. https://www.infoq.com/articles/java-virtual-threads/.

In real-world applications—such as social media web servers handling millions to billions of requests per month—efficiently performing large numbers of concurrent tasks is crucial. As we mentioned, platform threads have large memory overhead and map one-to-one to operating system threads, making applications that use them more complicated to scale for massive numbers of concurrent tasks. Virtual threads address this limitation by enabling millions of lightweight threads to run concurrently with minimal overhead. To demonstrate this, we'll build an example that executes large numbers of tasks using platform and virtual threads.

18.7.1 `Future` and `Callable` Interfaces

Each task in the next example returns a result. Interface `Runnable`'s `run` method returns `void`, so it cannot return a value. **Interface `Callable`** (package `java.util.concurrent`) fixes this limitation—its **`call` method** returns a value representing a task's result. `Callable`s

typically run concurrently with other `Runnable`s and `Callable`s. **`ExecutorService` method `submit`** executes its `Callable` argument and returns an object that implements the **`Future` interface** (package `java.util.concurrent`)—this represents the `Callable`'s result, which will be available at some point in the future. The **`Future` interface's `get` method** returns the task's result, possibly blocking the calling thread to wait until that result is available. The interface also provides methods that enable you to

- cancel a `Callable`'s execution,
- determine whether the `Callable` was canceled and
- determine whether the `Callable` completed its task.

18.7.2 Threading Approaches We'll Use

Figure 18.6 demonstrates virtual threads vs. platform threads handling large numbers of concurrent tasks that simulate blocking operations. The example submits tasks to three `ExecutorService` objects that manage the tasks with the following threading approaches:

- a fixed thread pool of 1000 pre-allocated platform threads that are shared among all the submitted tasks,
- a cached thread pool of platform threads that dynamically adjusts its number of threads based on demand and
- a separate virtual thread per task—because virtual threads use minimal resources and are easy to create in large numbers, you should not pool them.[21]

21. "JEP 444: Virtual Threads." Accessed February 9, 2025. https://openjdk.org/jeps/444.

We split the example into pieces for discussion purposes. Lines 3–9 import the classes and interfaces used in this example.

Click here to view code image

```
1  // Fig. 18.6: PlatformAndVirtualThreads.java
2  // Timing large numbers of platform vs. virtual threads.
3  import java.time.Duration;
4  import java.time.InstantSource;
5  import java.util.ArrayList;
6  import java.util.concurrent.ExecutorService;
7  import java.util.concurrent.Executors;
```

```
8  import java.util.concurrent.Future;
9  import java.util.stream.LongStream;
10
11 public class PlatformAndVirtualThreads {
```

Fig. 18.6 | Timing large numbers of platform vs. virtual threads.

18.7.3 Launching and Timing Large Numbers of Concurrent Tasks

Method `executeAndTimeThreads` (lines 13–48) manages the process of launching and timing the execution of large numbers of concurrent tasks. For demo purposes, each task in this example sleeps for one second to simulate blocking, then calculates and returns the square of a `long` value. We sum these squared values and display them in the sample outputs. Method `executeAndTimeThreads` requires three arguments:

- `tasks`—the number of concurrent tasks the method should execute,
- `threadType`—a `String` description of the threading approach used and
- `executor`—an `ExecutorService` implementation that creates the threads.

Click here to view code image

```
12    // execute and time threads; sum squares of 1 to number of tasks
13    private static void executeAndTimeThreads(int tasks,
14       String threadType, ExecutorService executor) throws Exception {
15       var clock = InstantSource.system(); // system clock for timing
16
17       // stores Futures for all the launched tasks
18       var futures = new ArrayList<Future<Long>>(tasks);
19
20       System.out.printf(
21          "Executing %d tasks on %s...%n", tasks, threadType);
22       var start = clock.instant();
23
24       // auto-closes the ExecutorService at the end of the try block;
25       // automatically waits for all tasks to complete
26       try (executor) {
27          // launch the number of tasks specified by tasks parameter
28          LongStream.rangeClosed(1, tasks).forEach(value ->
29             futures.add(executor.submit(() -> {
```

```
30                  Thread.sleep(Duration.ofSeconds(1)); // simulate blocking
31                  return value * value;
32               }))
33            );
34         }
35
36         // sum the results from the Futures
37         long sum = 0;
38         for (var future : futures) {
39            sum += future.get();
40         }
41
42         var end = clock.instant();
43
44         // display results
45         System.out.printf("Execution time: %.2f seconds%n",
46            Duration.between(start, end).toMillis() / 1000.0);
47         System.out.printf("Sum of squares 1-%d: %d%n%n", tasks, sum);
48      }
49
```

The method operates as follows:

- Line 15 gets the system clock for the timing operations.
- Line 18 creates an `ArrayList` to store `Future` objects for accessing each task's results.
- Lines 20–21 display the number of tasks being performed and the threading approach used in this call to `executeAndTimeThreads`.
- Line 22 records the system time before launching any tasks.
- Lines 26–34 are the `try`-with-resources statement in which we submit the tasks. We use a `LongStream` to count from `1` to `tasks`. For each `value`, we use the `executor` to `submit` a task—in this case, a lambda expression that implements the `Callable` interface, telling the thread to sleep for one second (to simulate a blocking), then returns the squared `value`. We store the `Future<Long>` returned by each call to `submit` in an `ArrayList`—we'll use this to get all the tasks' results later. When the stream finishes launching the tasks, control reaches the `try`-with-resources statement's closing brace, automatically closing the `executor` and waiting for the launched tasks to complete execution.
- When all the tasks complete, lines 37–40 total all the tasks' results to determine the sum of the squares of the values

from `1` through `tasks`.

- Line 42 records the system time after the tasks and final calculation complete execution.
- Lines 45–47 calculate and display the total execution time and final `sum`.

18.7.4 Testing the Threading Approaches

In the `main` method (lines 50–73):

- Lines 52–55 ensure the program was launched with one command-line argument. This value represents the number of threads to create, so we convert it to an `int` (line 58).
- Lines 61–63 invoke `executeAndTimeThreads`, passing the number of tasks to launch, a description of the kind of threading that will be used to execute tasks and an `ExecutorService` to launch the tasks. In this case, we use an `ExecutorService` that manages a fixed-size pool of 1000 platform threads. This thread pool can execute only 1000 tasks concurrently. Additional tasks must wait for executing tasks to complete before they can be assigned to threads in the pool.
- Lines 66–68 invoke `executeAndTimeThreads` with an `ExecutorService` that manages a cached thread pool, which dynamically adjusts its number of threads based on demand.
- Lines 71–72 invoke `executeAndTimeThreads` with an `ExecutorService` that creates a virtual thread per task.

Click here to view code image

```
50      public static void main(String[] args) throws Exception {
51         // check for proper command-line arguments
52         if (args.length != 1) {
53            System.out.println("Usage: PlatformAndVirtualThreads <count>");
54            System.exit(1);
55         }
56
57         // get number of threads to launch
58         final int TASKS = Integer.parseInt(args[0]);
59
60         // execute and time platform threads with fixed thread pool
61         executeAndTimeThreads(TASKS,
62            "Platform Threads (fixed pool of 1000 threads)",
```

```
            Executors.newFixedThreadPool(1000));

        // execute and time platform threads with cached thread pool
        executeAndTimeThreads(TASKS,
            "Platform Threads (cached thread pool)",
            Executors.newCachedThreadPool());

        // execute and time virtual threads
        executeAndTimeThreads(TASKS,
            "Virtual Threads", Executors.newVirtualThreadPerTaskExecutor());
    }
}
```

18.7.5 Performance Analysis

The following sample executions for 1,000, 5,000, 10,000 and 50,000 tasks were performed on a MacBook Pro with an M2 MAX processor with 96GB of RAM. Depending on your system configuration, you may not be able to execute as many tasks (possibly running out of memory for large numbers of tasks), or you may be able to execute even more than we demonstrate here. You can run the example with the following command

Click here to view code image

```
java PlatformAndVirtualThreads numberOfTasks
```

where *numberOfTasks* should be an integer representing the number of tasks to launch. The sample executions reveal interesting performance comparisons.

1,000 Tasks

There's little difference in execution times for the three threading approaches for 1,000 tasks—virtual threads had a slight advantage. In this case, the throughput using a fixed pool of platform threads is about 962 tasks-per-second (1000 / 1.04), using a cached pool of platform threads is about 971 (1000 / 1.03) and using virtual threads is about 980 tasks-per-second (1000 / 1.02):

Click here to view code image

```
Executing 1000 tasks on Platform Threads (fixed pool of 1000 threads)...
Execution time: 1.04 seconds
Sum of squares 1-1000: 333833500
```

```
Executing 1000 tasks on Platform Threads (cached thread pool)...
Execution time: 1.03 seconds
Sum of squares 1–1000: 333833500

Executing 1000 tasks on Virtual Threads...
Execution time: 1.02 seconds
Sum of squares 1–1000: 333833500
```

5,000 Tasks

In the sample execution for 5,000 tasks, the fixed-thread pool's throughput (986 tasks-per-second) is similar to that of 1,000—the number of tasks-per-second this pool can handle is limited by the fixed number of threads in the pool. There are not enough platform threads to handle all the tasks, so many tasks must wait until threads become available. The cached pool throughput increased to 3,759 tasks-per-second and the virtual threads throughput increased to 4,762 tasks-per-second. The cached pool of platform threads and the virtual threads scale better than the fixed pool of platform threads, with virtual threads again performing best.

Click here to view code image

```
Executing 5000 tasks on Platform Threads (fixed pool of 1000 threads)...
Execution time: 5.07 seconds
Sum of squares 1–5000: 41679167500

Executing 5000 tasks on Platform Threads (cached thread pool)...
Execution time: 1.33 seconds
Sum of squares 1–5000: 41679167500

Executing 5000 tasks on Virtual Threads...
Execution time: 1.05 seconds
Sum of squares 1–5000: 41679167500
```

10,000 Tasks

The Oracle virtual threads documentation indicates that "if your application never has 10,000 virtual threads or more, it is unlikely to benefit from virtual threads."[22] In the sample execution for 10,000 tasks, the benefits of virtual threads for large numbers of blocking tasks become more apparent. The fixed pool's throughput is 992 tasks-per-second—similar in scale to the throughput for 1,000 tasks and 5,000 tasks. The cached pool throughput increased to 7,518 tasks-per-second and the virtual threads throughput to

9,523 tasks-per-second. Again, the cached pool of platform threads and the virtual threads scale better, but the spread between the tasks-per-second of virtual threads and the cached pool of platform threads increased.

22. "Virtual Threads." Accessed February 9, 2025. https://docs.oracle.com/en/java/javase/23/core/virtual-threads.html.

Click here to view code image

```
Executing 10000 tasks on Platform Threads (fixed pool of 1000 threads)...
Execution time: 10.08 seconds
Sum of squares 1-10000: 333383335000

Executing 10000 tasks on Platform Threads (cached thread pool)...
Execution time: 1.93 seconds
Sum of squares 1-10000: 333383335000

Executing 10000 tasks on Virtual Threads...
Execution time: 1.07 seconds
Sum of squares 1-10000: 333383335000
```

50,000 Tasks

In the sample execution for 50,000 tasks, the benefits of virtual threads for large numbers of blocking tasks become crystal clear. The fixed pool's throughput (994 tasks-per-second) is similar to prior sample executions. The cached pool's throughput actually decreased in this execution to 5,507 tasks-per-second, but **the virtual threads' throughput increased to 34,468 tasks-per-second**.

Click here to view code image

```
Executing 50000 tasks on Platform Threads (fixed pool of 1000 threads)...
Execution time: 50.26 seconds
Sum of squares 1-50000: 41667916675000

Executing 50000 tasks on Platform Threads (cached thread pool)...
Execution time: 9.08 seconds
Sum of squares 1-50000: 41667916675000

Executing 50000 tasks on Virtual Threads...
Execution time: 1.54 seconds
Sum of squares 1-50000: 41667916675000
```

The sample outputs indicate that virtual threads consistently

outperform platform threads for highly concurrent tasks that spend most of their time blocked, especially as the number of threads increases. This enables applications with large numbers of blocking tasks to scale without the limitations of traditional platform threads.

Generative AI

1 Prompt genAIs for key virtual thread use-cases in desktop applications.

2 Prompt genAIs, asking why platform threads are better than virtual threads for compute-intensive (CPU-bound) tasks.

3 Prompt genAIs for virtual thread best practices.

4 Prompt genAIs for recommendations on choosing the appropriate number of threads to optimize performance when using thread pools.

5 Prompt genAIs to explain how blocking affects application performance.

18.8 Structured Concurrency and Scoped Values

This section introduces Project Loom's two other key features—structured concurrency and scoped values:

- **Structured concurrency** simplifies task management by enabling developers to treat groups of virtual threads performing related tasks as a single unit and program with them in a simple statement-by-statement sequential programming style. This makes the code more readable and easier to maintain, modify and debug. It also avoids more complex asynchronous programming techniques, such as reactive programming.[23] Structured concurrency also simplifies resource management and error handling because those sequential statements can be placed in familiar `try` and

`try`-with-resources statements.

- **Scoped values** enable a task and its subtasks to share immutable data in a structured concurrency scope. They fix problems associated with `ThreadLocal` variables, introduced in Java almost three decades ago.[24] Scoped values become implicit arguments to every method called from that structured concurrency scope.

23. The details of reactive programming are beyond this book's scope. For some details on how virtual threads are better than reactive techniques, read: Goetz, Brian. "Virtual Threads: New Foundations for High-Scale Java Applications." InfoQ, November 2, 2022. Accessed February 9, 2025. https://www.infoq.com/articles/java-virtual-threads/.

24. The details of `ThreadLocal` variables are beyond this book's scope. For more information on how scoped values are better than `ThreadLocal` variables, visit https://openjdk.org/jeps/487 and see Generative AI Exercise 18.2.

Example Overview

To demonstrate structured concurrency and scoped values, let's modify Section 11.6's OpenWeatherMap web-services example that returns the current weather for a specified city. Recall that virtual threads are primarily used for large numbers of concurrent tasks that spend most of their time blocking—a web-service request blocks until its results are available or the request fails. Let's assume we'd like to build an application in which each user can submit a request that specifies a city for which the user would like the current weather and the forecast for the next few days—these are separate OpenWeatherMap web-service requests. As in Section 11.6, we'll use the Jackson library to parse the JSON responses for convenient access to the weather data.

For demo purposes, we'll build this as a command-line app that simulates "requests" (the individual cities) as command-line arguments. We'll use **thread-per-request handling**, launching a virtual thread for each request. That thread, in turn, will launch two subtasks in separate virtual threads so they may proceed concurrently:

- one to invoke the OpenWeatherMap web service that returns the city's current weather and
- one to invoke the OpenWeatherMap web service that returns

the city's five-day forecast in three-hour increments.

If a subtask fails, we'll consider that an error for which we'd like the overall request and its subtasks to terminate immediately—structured concurrency can do this automatically. In addition, the request and its subtasks require a shared piece of data—the city for which to get the weather. This information does not change for a particular request's duration, so we'll use a scoped value to share that immutable data between the virtual thread handling the request and its subtask virtual threads.

Compiling and Running This Example

Structured concurrency and scoped values were preview features at the time of this writing and some changes are expected before they become standard Java features. We'll post updates on the book's website at

```
https://deitel.com/javafp5
```

Until they become standard, you must enable them when compiling and running the code. This example also uses the Jackson open-source library located in the `libs` subfolder with this chapter's examples. To compile the program on macOS, enter the command:

Click here to view code image

```
javac --enable-preview -source 23 -cp ".:../libs/*" WeatherApp.java
```

where

- The **`--enable-preview` option** indicates that we use preview features.
- The **`-source 23` option** indicates that we're compiling with Java 23.

These options can be removed once structured concurrency and scoped values are official Java features. On Windows, replace the forward slashes (`/`) with backslashes (`\`) and the colon (`:`) with a semicolon (`;`).

To run the code on macOS, enter the following command

Click here to view code image

```
java --enable-preview -cp ".:../libs/*" WeatherApp Boston \
   "New Orleans" "Los Angeles"
```

On Windows, replace the backslash (`\`) with a caret (`^`) if you're using a Command Prompt window or a backtick (`` ` ``) if you're using a PowerShell window. Multiword city names must be enclosed in quotes (`""`); otherwise, each word is treated as a separate city.

`import` Statements

Lines 4–20 import the classes and interfaces used in this example, with lines 6 and 19 importing the `ScopedValue` and `StructuredTaskScope` classes, respectively.

Click here to view code image

```
1  // Fig. 18.7: WeatherApp.java
2  // Fetch a city's weather using OpenWeatherMap API with
3  // structured concurrency and scoped values.
4  import com.fasterxml.jackson.annotation.JsonIgnoreProperties;
5  import com.fasterxml.jackson.databind.ObjectMapper;
6  import java.lang.ScopedValue;
7  import java.net.URI;
8  import java.net.URLEncoder;
9  import java.net.http.HttpClient;
10 import java.net.http.HttpRequest;
11 import java.net.http.HttpResponse;
12 import java.nio.charset.StandardCharsets;
13 import java.time.Instant;
14 import java.time.ZoneId;
15 import java.time.ZoneOffset;
16 import java.time.format.DateTimeFormatter;
17 import java.util.List;
18 import java.util.concurrent.Executors;
19 import java.util.concurrent.StructuredTaskScope;
20 import java.util.stream.Collectors;
21
```

Fig. 18.7 | Fetch a city's weather using OpenWeatherMap API with structured concurrency and scoped values.

`record` Classes Used for Deserializing OpenWeatherMap JSON Responses

Lines 23–31 contain the `record` classes we'll use to store portions of the JSON response for a city's current weather, and lines 34–38 contain the `record` classes we'll use to store portions of the JSON response for a city's weather forecast. Read Section 11.6's discussion for more details. You can study a sample complete JSON response for the current weather at

Click here to view code image

```
https://openweathermap.org/current
```

and for the weather forecast at

Click here to view code image

```
https://openweathermap.org/forecast5
```

We use less of the OpenWeatherMap current weather data for this example than in Section 11.6's example. For each day in the forecast data, we use the date, weather description and the minimum and maximum temperatures.

Click here to view code image

```
22  // record classes for current weather JSON mapping
23  @JsonIgnoreProperties(ignoreUnknown = true)
24  record WeatherData(Weather[] weather, Main main, long dt,
25     int timezone, String name) {}
26
27  @JsonIgnoreProperties(ignoreUnknown = true)
28  record Weather(String main, String description) {}
29
30  @JsonIgnoreProperties(ignoreUnknown = true)
31  record Main(double temp) {}
32
33  // record classes for forecast JSON mapping
34  @JsonIgnoreProperties(ignoreUnknown = true)
35  record ForecastResponse(List<ForecastEntry> list) {}
36
37  @JsonIgnoreProperties(ignoreUnknown = true)
38  record ForecastEntry(long dt, Main temp, Weather[] weather) {}
39
```

`static` Class Variables

Lines 42–43 load the OpenWeatherMap API key—you originally obtained this key and stored it in an environment variable in Section 11.6. Lines 46–47 define an `HttpClient` object once, so we do not need to re-create it for every web-service request. Lines 50–51 define a **`ScopedValue`** that stores a `String` and initialize it by calling **`ScopedValue`**'s **`static`** **method** **`newInstance`**, which returns a `ScopedValue` object. We'll specify this object's city value as we prepare to launch the task that retrieves a given city's weather.

Click here to view code image

```
40  public class WeatherApp {
41     // load OpenWeatherMap API key
42     private static final String API_KEY =
43        System.getenv("OPENWEATHERMAP_API_KEY");
44
45     // HttpClient for making web-service requests
46     private static final HttpClient HTTP_CLIENT =
47        HttpClient.newHttpClient();
48
49     // used to launch each group of requests for a specified city
50     private static final ScopedValue<String> CITY =
51        ScopedValue.newInstance();
52
```

`main` Method

The `main` method (lines 53–83) simulates a server that implements thread-per-request handling. The `try`-with-resources statement gets an `ExecutorService` for launching virtual threads (line 61). After launching a virtual thread for each request (i.e., city), this statement closes the `ExecutorService` and waits for all the launched virtual threads to complete. The loop in lines 62–81 uses the `executor` to launch a virtual thread for each city. The lambda expression in lines 66–79 defines a `Runnable` task:

- Line 66 first calls the **`ScopedValue`'s `where` method** to set the `ScopedValue`'s value to the current request's city for which to get the weather. This returns an object of **class `ScopedValue.Carrier`** containing the unique scoped value for the task we're about to launch.
- Next, line 66 calls the **`ScopedValue.Carrier` object's `run` method** to execute the `Runnable` task defined by the lambda expression in lines 68–78. The `run` method provides the scoped value to that task. We'll use structured concurrency in the `getWeather` method (called at line 71) to manage the task and its subtasks.
- Lines 70–71 display the `String` returned by `getWeather`.

Click here to view code image

```
53     public static void main(String[] args) throws Exception {
54        // check for proper command-line arguments
55        if (args.length == 0) {
56           System.out.println("Usage: WeatherApp <city1> <city2> ...");
```

```
57            System.exit(1);
58         }
59
60         // simulate a server doing thread-per-request handling
61         try (var executor = Executors.newVirtualThreadPerTaskExecutor()) {
62            for (String city : args) {
63               executor.submit(
64                  // set the ScopedValue's value then run a task; the task
65                  // and its subtasks will have access to the ScopedValue
66                  () -> ScopedValue.where(CITY, city).run(
67                     // task: get and print the weather info for the city
68                     () -> {
69                        try {
70                           System.out.printf("%n%nWeather for %s:\n%s\n",
71                              city, getWeather());
72                        }
73                        catch (Exception ex) {
74                           System.err.printf(
75                              "Error fetching weather for %s: %s\n",
76                              city, ex.getMessage());
77                        }
78                     } // end nested lambda expression
79                  ) // end call to ScopedValue's run method
80               ); // end call to the executor object's submit method
81            } // end for loop
82         } // end try-with-resources statement
83      }
84
```

`getWeather` Method

The `getWeather` method (lines 86–116) uses the `CITY` **`ScopedValue`'s `get` method** (line 89) to get the city to use in the task and subtasks launched from `getWeather`. The `get` method automatically accesses the scoped value in the `ScopedValue.Carrier` object associated with the current task. Lines 92–97 form the URLs for the two `OpenWeatherMap` web-service requests we'll perform. Next, the `try`-with-resources statement (lines 100–115) creates a new **`StructuredTaskScope`** (line 100). For this example, we consider the task in method `getWeather` successful only if both web-service requests complete successfully. The expression in line 100

Click here to view code image

```
new StructuredTaskScope.ShutdownOnFailure()
```

creates a `StructuredTaskScope` that shuts down on failure. If a

subtask launched in this scope fails, the other subtasks will be interrupted so they can shut down. Lines 101–104 call **`StructuredTaskScope`'s `fork` method** to launch virtual threads that execute `Callable` tasks—implemented by the lambdas in lines 102 and 104. Each lambda calls our generic `requestData` method (discussed momentarily), passing the URL for the web-service request and the `Class` object (`WeatherData.class` or `ForecastResponse.class`) the Jackson library will use when extracting the data items we need from the JSON responses. Method `requestData` returns an object of the appropriate `record` class (`WeatherData` or `ForecastResponse`) for each request. Each `fork` call returns an object that implements the **`StructuredTaskScope.Subtask` interface**, which we'll use to access the subtasks' results when they become available.

Click here to view code image

```
85      // use structured concurrency to request current weather and forecast
86      public static String getWeather() throws Exception {
87         // URL encode the city parameter
88         String encodedCity =
89            URLEncoder.encode(CITY.get(), StandardCharsets.UTF_8);
90
91         // construct API URLs
92         String currentWeatherUrl = String.format(
93            "https://api.openweathermap.org/data/2.5/weather?q=%s" +
94            "&units=metric&appid=%s", encodedCity, API_KEY);
95         String forecastUrl = String.format(
96            "https://api.openweathermap.org/data/2.5/forecast?q=%s" +
97            "&units=metric&appid=%s", encodedCity, API_KEY);
98
99         // use StructuredTaskScope to manage concurrent tasks
100        try (var scope = new StructuredTaskScope.ShutdownOnFailure()) {
101           var currentWeatherTask = scope.fork(
102              () -> requestData(currentWeatherUrl, WeatherData.class));
103           var forecastTask = scope.fork(
104              () -> requestData(forecastUrl, ForecastResponse.class));
105
106           scope.join(); // waits for responses
107           scope.throwIfFailed(); // throws an exception if a task fails
108
109           // tasks completed successfully, retrieve results
110           WeatherData weatherData = currentWeatherTask.get();
111           ForecastResponse forecastResponse = forecastTask.get();
112
113           // format and return output
114           return formatWeatherOutput(weatherData, forecastResponse);
115        }
```

```
116     }
117
```

Line 106 calls **`StructuredTaskScope`'s `join` method** to wait for the two web-service requests to complete. If either fails, line 107 calls **`StructuredTaskScope`'s `throwIfFailed` method** to ensure the task's exception can be handled. In this example, we handle it in `main` rather than in a `catch` handler following the `try`-with-resources statement. If both tasks complete successfully, lines 110–111 get the `WeatherData` or `ForecastResponse` objects by calling **`StructuredTaskScope.Subtask` method `get`**. Finally, line 114 calls method `formatWeatherOutput` to produce the weather report and returns the resulting `String` to `main` for display at lines 70–71.

`requestData` Method

The `requestData` method (lines 119–138) uses the techniques presented in Section 11.6 to

- build an `HttpRequest` (lines 121–122),
- submit the request via an `HttpClient` and wait for the response (lines 125–126),
- check whether the response was successful (lines 123–133), and
- deserialize the JSON data into a `record` object (line 137).

Click here to view code image

```
118     // generic method to retrieve JSON data from OpenWeatherMap API
119     private static <T> T requestData(String url, Class<T> type)
120        throws Exception {
121        // build the request to the OpenWeatherMap web service
122        HttpRequest request = HttpRequest.newBuilder()
123           .uri(URI.create(url)).GET().build();
124
125        // send the HTTP request and get the response
126        HttpResponse<String> response = HTTP_CLIENT.send(request,
127           HttpResponse.BodyHandlers.ofString());
128
129        // check if the request was successful; otherwise, throw Exception
130        if (response.statusCode() != 200) {
131           throw new Exception(String.format(
132              "Error: HTTP status %d\n", response.statusCode()));
133        }
134
```

```
135         // use Jackson library to deserialize the JSON response
136         // into the format specified by type
137         return new ObjectMapper().readValue(response.body(), type);
138      }
139
```

`formatWeatherOutput` Method

The `formatWeatherOutput` method (lines 141–174) receives the record objects containing the data from the two web-service requests then creates a formatted weather report:

- Lines 145–146 get the current temperature and convert it from Celsius to Fahrenheit.
- Lines 149–152 create a `DateTimeFormatter` configured to display the date and time in the format `yyyy-MM-dd HH:mm`.
- Lines 155–164 use stream processing to convert the first 16 items in the `ForecastResponse`'s `List<ForecastEntry>` into formatted text containing the date, time, weather description and temperature in both Celsius and Fahrenheit.
- Lines 166–173 return the result of calling `String` method `formatted` on a format `String` implemented as a text block, inserting the city's name, the current day's weather description, the current Celsius and Fahrenheit temperatures and the `forecastInfo` from lines 155–164.

Click here to view code image

```
140      // formats weather output for display
141      private static String formatWeatherOutput(
142         WeatherData weatherData, ForecastResponse forecastResponse) {
143
144         // current temperature
145         double tempC = weatherData.main().temp();
146         double tempF = (tempC * 9 / 5) + 32;
147
148         // date formatter
149         DateTimeFormatter dateFormatter =
150            DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm")
151               .withZone(ZoneId.ofOffset("UTC",
152                  ZoneOffset.ofTotalSeconds(weatherData.timezone())));
153
154         // 7-day forecast of weather descriptions with min/max temperatures
155         String forecastInfo = forecastResponse.list().stream().limit(16)
156            .map(f -> {
```

```
            double tempCForecast = f.main().temp();
            double tempFForecast = (tempCForecast * 9 / 5) + 32;
            return String.format(
               "%s: %s; temp: %.1f C (%.1f F)",
               dateFormatter.format(Instant.ofEpochSecond(f.dt())),
               f.weather()[0].main(), tempCForecast, tempFForecast);
         })
         .collect(Collectors.joining("\n"));

      return """
         %s Weather: %s
         Temperature: %.1f C (%.1f F)

         Forecast:
         %s""".formatted(weatherData.name(),
            weatherData.weather()[0].description(),
            tempC, tempF, forecastInfo);
   }
}
```

Sample Output

The following sample output contains the result of executing this application with Boston, New Orleans and Los Angeles as the command-line arguments. Note that the tasks completed and their results were displayed in a different order than when we launched the tasks. Remember that we cannot predict the order in which the virtual threads will execute. In addition, there are network and processing delays as our application sends the request to the web service, the web service formulates its response, and the web service returns its response to our application.

Click here to view code image

```
Weather for New Orleans:
New Orleans Weather: scattered clouds
Temperature: 13.6 C (56.5 F)

Forecast:
2025-03-06 21:00: Clouds; temp: 13.3 C (56.0 F)
2025-03-07 00:00: Clouds; temp: 12.3 C (54.2 F)
2025-03-07 03:00: Clouds; temp: 11.5 C (52.8 F)
2025-03-07 06:00: Clouds; temp: 11.6 C (52.8 F)
2025-03-07 09:00: Clouds; temp: 17.3 C (63.1 F)
2025-03-07 12:00: Clouds; temp: 22.7 C (72.8 F)
...

Weather for Los Angeles:
```

```
Los Angeles Weather: moderate rain
Temperature: 8.9 C (47.9 F)

Forecast:
2025-03-06 19:00: Rain; temp: 9.0 C (48.2 F)
2025-03-06 22:00: Clouds; temp: 9.4 C (48.9 F)
2025-03-07 01:00: Clear; temp: 9.3 C (48.7 F)
2025-03-07 04:00: Clear; temp: 8.4 C (47.1 F)
2025-03-07 07:00: Clear; temp: 8.7 C (47.6 F)
2025-03-07 10:00: Clear; temp: 12.8 C (55.0 F)
...

Weather for Boston:
Boston Weather: moderate rain
Temperature: 6.0 C (42.9 F)

Forecast:
2025-03-06 22:00: Clouds; temp: 4.9 C (40.8 F)
2025-03-07 01:00: Clouds; temp: 1.4 C (34.5 F)
2025-03-07 04:00: Clouds; temp: -2.9 C (26.8 F)
2025-03-07 07:00: Clouds; temp: -2.0 C (28.3 F)
2025-03-07 10:00: Clouds; temp: 1.7 C (35.1 F)
2025-03-07 13:00: Clouds; temp: 4.7 C (40.5 F)
...
```

Generative AI

1 This section's example demonstrated a `StructuredTaskScope` configured to shut down on failure. You can also specify a `StructuredTaskScope` configured to shut down as soon as any of its subtasks completes successfully. Prompt genAIs, asking for examples of why you might use this feature and a simple code example demonstrating its use.

2 Prompt genAIs, asking for examples of why you might use nested `StructuredTask-Scope`s and a simple code example demonstrating this feature.

3 Prompt genAIs, asking for cases in which tasks might require more than one `ScopedValue` and a simple code example demonstrating how to provide them to a given task.

4 Prompt genAIs, asking why you must create a `ScopedValue` via a `static` method rather than a constructor.

5 Prompt genAIs, asking how structured concurrency simplifies

error handling.

6 Prompt genAIs for structured concurrency best practices.

7 Prompt genAIs for `ScopedValue` best practices.

18.9 Thread Synchronization Overview

When multiple threads share an object, and it's modified by one or more of them, indeterminate results may occur unless access to the shared object is managed correctly:

- If one thread is updating a shared object and another thread tries to update it at the same time, it's uncertain which thread's update takes effect.
- Similarly, if one thread is updating a shared object and another thread tries to read it at the same time, it's uncertain whether the reading thread will read the old value that's no longer valid (known as a **stale value**) or the new one.

In such cases, the program's behavior cannot be trusted. Sometimes the program will produce the correct results, and sometimes it won't, and there won't be any indication that the shared object was manipulated incorrectly.

The problem can be solved by giving only one thread at a time exclusive access to code that manipulates the shared object. During that time, other threads desiring to access the object are kept waiting. When the thread with exclusive access finishes manipulating the object, one of the waiting threads can proceed. This process, called **thread synchronization**, coordinates access to shared data by multiple concurrent threads. By synchronizing threads in this manner, you can ensure that each thread accessing a shared object excludes all other threads from doing so simultaneously—this is called **mutual exclusion**.

Java provides extensive mechanisms designed to ensure mutually exclusive access to shared mutable data, from low-level capabilities that enable developers to manage mutual exclusion to high-level ones in which the Java APIs handle mutual exclusion for you. **One of the challenges of multithreaded programming is spotting the**

errors—they may occur so infrequently and unpredictably that a broken program does not produce incorrect results during testing, creating the illusion that the program is correct. For this reason, most developers should work with the predefined concurrent collections that handle thread synchronization for you. These collections are designed by Java concurrency experts to work correctly and perform optimally in applications that share mutable data.

Immutable Data

Thread synchronization is necessary only for **shared mutable data**—that is, data that may change during its lifetime. With **shared immutable data** that will not change, it's impossible for a thread to read old or incorrect values due to another thread's manipulation of that data.

When you share immutable data across threads, declare the corresponding data fields `final` to indicate that the variable's values will not change after they're initialized. This prevents accidental modification of the shared data, which could compromise thread safety. **Labeling object references as `final` indicates that the reference will not change, but it does not guarantee that the referenced object is immutable—this depends entirely on the object's properties.** But, it's still good practice to mark references that will not change as `final`.

As a good practice, declare data fields you do not expect to change as `final`. Primitive variables that are declared as `final` can safely be shared across threads. An object reference declared as `final` ensures that the object it refers to will be fully constructed and initialized before it's used by the program, preventing the reference from pointing to another object.

Generative AI

1 Prompt genAIs, asking why immutable data is preferred for sharing information among threads in a Java application.

2 Prompt genAIs, asking for the kinds of concurrency issues you might experience in Java applications.

18.10 Producer/Consumer Relationship

A common scenario in which multiple threads access shared mutable data is in a **producer/consumer relationship**:

- The **producer** portion of a program generates data and stores it in a shared object, and
- the **consumer** portion of the program reads data from the shared object and processes it.

Examples of Producer/Consumer Relationship

One example of a producer/consumer relationship is **print spooling**. Although a printer might not be available when you want to print from an application (i.e., the producer), you can still "complete" the print task, as the data is temporarily placed on disk until the printer becomes available. Similarly, when the printer (i.e., a consumer) is available, it doesn't have to wait until a current user wants to print. The spooled print jobs can print when the printer becomes available. Another example of the producer/consumer relationship is an application that backs up data to a storage device—such as a solid-state drive (SSD)—by placing data in a fixed-size buffer, which is emptied as the storage device writes the data.

Generative AI

1 Prompt genAIs for real-world Java examples of producer/consumer relationships.

18.10.1 Synchronization and State Dependence

In a multithreaded producer/consumer relationship, a **producer thread** generates data and places it in a shared object called a buffer or a queue. A **consumer thread** reads data from the buffer. This relationship requires synchronization to ensure that values are produced and consumed properly to prevent data corruption:

- Data can be lost or garbled if the producer places new data into the shared buffer before the consumer reads the previous data.
- Also, data can be incorrectly duplicated if the consumer consumes data again before the producer produces the next value.

Operations on the buffer data shared by a producer and consumer thread are also **state-dependent**. As an example of state dependence, consider an electric vehicle. You can't drive an electric vehicle if its battery is not charged, and you can't charge an electric vehicle's battery more, if it's already fully charged. Similarly, a producer and consumer should proceed only if the buffer is in the correct state:

- If the buffer is in a not-full state, the producer may produce.
- If the buffer is in a not-empty state, the consumer may consume.

All operations that access the buffer must use synchronization to ensure that data is written to the buffer or read from the buffer only if the buffer is in the proper state:

- If the producer attempts to write a new value into a full buffer, the producer thread must wait until there's space to write the value.
- If a consumer thread attempts to read a value from an empty buffer, the consumer must wait for new data to become available.

18.10.2 Class `ArrayBlockingQueue`

A reliable way to synchronize producers and consumers is to use classes that implement the synchronization for you. The `java.util.concurrent` package's **`ArrayBlockingQueue` class** defines a fully implemented, thread-safe buffer class that implements the **interface `BlockingQueue`**. Among its methods, this interface declares:

- **method `put`** to place an element at the tail of a `BlockingQueue`, waiting if the queue is full, and
- **method `take`** to remove an element from the head of the

`BlockingQueue`, waiting if the queue is empty.

These methods make class `ArrayBlockingQueue` a good choice for implementing a shared buffer. Because method `put` blocks until there's room in the buffer to write data, and method `take` blocks until there's new data to read:

- the producer must produce a value first,
- the consumer correctly consumes only after the producer writes a value, and
- the producer correctly produces the next value (after the first) only after the consumer reads the previous (or first) value.

`ArrayBlockingQueue` stores the shared mutable data in an array, the size of which is specified as `ArrayBlockingQueue`'s constructor argument. An `ArrayBlockingQueue` is fixed in size and will not expand to accommodate extra elements.

18.10.3 Overview of the Producer/Consumer Example

Our next example (Figs. 18.8–18.10) implements a producer/consumer relationship with proper synchronization provided by a one-element `ArrayBlockingQueue`. The producer thread writes the numbers 1 through 10 into the shared `ArrayBlockingQueue`. The consumer thread reads this data from the shared `ArrayBlockingQueue`. Each value the producer thread writes must be consumed by the consumer thread exactly once. To help confirm this,

- the producer thread will output each value it writes (produces) and add the values to a running total, and
- the consumer thread will output each value it reads (consumes) and add the values to a running total.

If the producer writes each value from 1–10 exactly once and the consumer reads each value from 1–10 exactly once, each thread's total of the values should be 55.

To further emphasize that the `ArrayBlockingQueue` provides the synchronization, the producer and consumer threads each sleep for random intervals of up to 3,000 milliseconds between attempting to

write or read a value, respectively. As a result, we do not know when the producer thread will attempt to write a new value into the shared buffer or when the consumer thread will attempt to read what's in the shared buffer. The program consists of the following interface and classes:

- interface `Buffer` (Fig. 18.8),
- class `BlockingBuffer` (Fig. 18.9) and
- class `BlockingBufferTest` (Fig. 18.10).

18.10.4 Interface `Buffer`

Interface `Buffer` (Fig. 18.8) declares methods `blockingPut` (line 5) and `blockingGet` (line 8) that our `BlockingBuffer` class implements for a producer thread to place a value in the buffer and a consumer thread to retrieve a value from it, respectively. Methods `blockingPut` and `blockingGet` will call `ArrayBlockingQueue` methods that throw `InterruptedException`s. They will also output text for pedagogic purposes, which is why this example declares interface `Buffer` rather than calling `ArrayBlockingQueue` methods directly. Interface `Buffer`'s methods list `InterruptedException` in their `throws` clauses so the exceptions can be returned to the producer or consumer threads calling `Buffer`'s methods.

Click here to view code image

```
1  // Fig. 18.8: Buffer.java
2  // Buffer interface specifies methods called by Producer and Consumer.
3  public interface Buffer {
4     // place int value into Buffer
5     public void blockingPut(int value) throws InterruptedException;
6
7     // return int value from Buffer
8     public int blockingGet() throws InterruptedException;
9  }
```

Fig. 18.8 | Buffer interface specifies methods called by `Producer` and `Consumer`.

18.10.5 Class `BlockingBuffer`

Class `BlockingBuffer` (Fig. 18.9) implements the `Buffer` interface and uses an object of class `ArrayBlockingQueue` to store one

`Integer` (lines 7–8) and manage mutually exclusive access to it—we'll say more about an `ArrayBlockingQueue`'s size at the end of this section. Because our `BlockingBuffer` class uses the thread-safe `ArrayBlockingQueue` class to manage all of its shared state (the shared buffer), `BlockingBuffer` is itself thread-safe, even though we have not implemented the synchronization ourselves.

Click here to view code image

```
1  // Fig. 18.9: BlockingBuffer.java
2  // Creating a synchronized buffer using an ArrayBlockingQueue.
3  import java.util.concurrent.ArrayBlockingQueue;
4
5  public class BlockingBuffer implements Buffer {
6     // BlockingBuffer uses an ArrayBlockingQueue to manage synchronization
7     private final ArrayBlockingQueue<Integer> buffer =
8        new ArrayBlockingQueue<>(1);
9
10    // place value into buffer
11    @Override
12    public void blockingPut(int value) throws InterruptedException {
13       buffer.put(value); // place value in buffer
14       System.out.printf("%s%2d\t%s%d%n", "Producer writes ", value,
15          "Buffer cells occupied: ", buffer.size());
16    }
17
18    // return value from buffer
19    @Override
20    public int blockingGet() throws InterruptedException {
21       int readValue = buffer.take(); // remove value from buffer
22       System.out.printf("%s %2d\t%s%d%n", "Consumer reads ",
23          readValue, "Buffer cells occupied: ", buffer.size());
24
25       return readValue;
26    }
27 }
```

Fig. 18.9 | Creating a synchronized buffer using an `ArrayBlockingQueue`.

Line 13 in method `blockingPut` calls the `ArrayBlockingQueue` object's `put` method. If the `ArrayBlockingQueue` is full, the producer thread waits at line 13 until there's room in the `ArrayBlockingQueue` to write the `value`. Line 21 in method `blockingGet` calls the `ArrayBlockingQueue` object's `take` method. If the `ArrayBlockingQueue` is empty, the consumer thread waits at line 21 until there's an element in the `ArrayBlockingQueue` to read. Lines 14–15 and 22–23 use the **`ArrayBlockingQueue` `size` method**

to display the `ArrayBlockingQueue`'s current number of elements. In this example, when the size is `1`, the buffer is full, and when it's `0`, the buffer is empty.

The `ArrayBlockingQueue` approach used in this example demonstrates encapsulated synchronization—**the producer and consumer threads that call `blockingPut` and `blockingGet` to access the shared `ArrayBlockingQueue` (i.e., the buffer) are unaware that their writes and reads are synchronized.** This is handled entirely by the `ArrayBlockingQueue` methods `put` and `take`.

18.10.6 Class `BlockingBufferTest`

Class `BlockingBufferTest`'s `main` method (Fig. 18.10) launches two virtual threads to execute a producer task that writes values into a shared `BlockingBuffer` and a consumer task that reads values from the same `BlockingBuffer`. We split class `BlockingBufferTest` into parts for discussion purposes. The tasks in this example are not CPU-intensive and block if they cannot perform their tasks, so line 10 creates an `ExecutorService` for virtual threads. Line 12 creates the shared `BlockingBuffer` object that both the producer and consumer tasks will access.

Click here to view code image

```
1  // Fig. 18.10: BlockingBufferTest.java
2  // Two threads manipulating a blocking buffer that properly
3  // implements the producer/consumer relationship.
4  import java.util.concurrent.Executors;
5  import java.util.concurrent.ThreadLocalRandom;
6
7  public class BlockingBufferTest {
8     public static void main(String[] args) throws InterruptedException {
9        // virtual-thread-per-task executor
10       try (var executor = Executors.newVirtualThreadPerTaskExecutor()) {
11          // create BlockingBuffer to store ints
12          final Buffer buffer = new BlockingBuffer();
13
```

Fig. 18.10 | Two threads manipulating a blocking buffer that properly implements the producer/consumer relationship.

Producer Task

Lines 15–31 define the producer's `Runnable` task as a lambda

expression and launch its execution in a virtual thread. Each iteration of its loop (lines 18–27) invokes `Thread` method `sleep` (line 20) to place the producer's virtual thread to sleep for a random time interval. When the thread awakens and begins executing again, line 21 passes the control variable `count`'s value to the `buffer` object's `blockingPut` method to set the shared buffer's value. Line 22 keeps a total of all the values produced so far so we can output that total when the producer finishes executing. When the loop completes, lines 29–30 output a message indicating that the producer finished executing and the total of all the values produced. The producer virtual thread terminates when the flow of control reaches the end of the lambda that defines its `Runnable`'s `run` method.

Click here to view code image

```
14          // producer task writes 1-10 into buffer
15          executor.execute(() -> {
16             int sum = 0;
17
18             for (int count = 1; count <= 10; count++) {
19                try {
20                   Thread.sleep(ThreadLocalRandom.current().nextInt(3000));
21                   buffer.blockingPut(count); // write value into buffer
22                   sum += count; // keep running total of values written
23                }
24                catch (InterruptedException exception) {
25                   Thread.currentThread().interrupt();
26                }
27             }
28
29             System.out.printf(
30                "Producer done: Produced values totaling %d%n", sum);
31          });
32
```

Consumer Task

Lines 34–49 define the consumer's `Runnable` task and launch its execution in a virtual thread, enabling it to run concurrently with the producer's virtual thread. Lines 37–45 iterate 10 times. Each iteration invokes `Thread` method `sleep` (line 39) to put the consumer's virtual thread to sleep for a random time interval. Next, line 40 calls the `buffer`'s `blockingGet` method to retrieve the value in the shared buffer, then adds the value to variable `sum`. When the loop completes, lines 47–48 indicate that the consumer finished and

display the sum of the consumed values, which should match the sum of the values the producer produced. The consumer virtual thread terminates when the flow of control reaches the end of the lambda that defines its `Runnable`'s `run` method. The program terminates once both virtual threads in this program complete their tasks.

Click here to view code image

```
            // consumer task reads values from buffer
            executor.execute(() -> {
               int sum = 0;

               for (int count = 1; count <= 10; count++) {
                  try {
                     Thread.sleep(ThreadLocalRandom.current().nextInt(3000));
                     sum += buffer.blockingGet(); // read value from buffer
                  }
                  catch (InterruptedException exception) {
                     Thread.currentThread().interrupt();
                  }
               }

               System.out.printf(
                  "%nConsumer done: Read values totaling %d%n", sum);
            });
         }
      }
   }
```

Sample Output

As you execute the program, you'll notice some pauses between the writes and reads:

- Sometimes, these are the result of the program's threads sleeping.
- Sometimes, they result from the producer or consumer thread having to wait its turn to access the `BlockingBuffer`'s `ArrayBlockingQueue`.

The `ArrayBlockingQueue` is implemented to correctly manage access to its data from concurrent threads, so the program should operate correctly each time with

- the producer writing each value from 1 to 10 once,

- the consumer reading each value once and
- both displaying the same total for the sum of the values from 1 to 10.

Click here to view code image

```
Producer writes   1      Buffer cells occupied: 1
Consumer reads    1      Buffer cells occupied: 0
Producer writes   2      Buffer cells occupied: 1
Consumer reads    2      Buffer cells occupied: 0
Producer writes   3      Buffer cells occupied: 1
Consumer reads    3      Buffer cells occupied: 0
Producer writes   4      Buffer cells occupied: 1
Consumer reads    4      Buffer cells occupied: 0
Producer writes   5      Buffer cells occupied: 1
Consumer reads    5      Buffer cells occupied: 0
Producer writes   6      Buffer cells occupied: 1
Consumer reads    6      Buffer cells occupied: 0
Producer writes   7      Buffer cells occupied: 1
Consumer reads    7      Buffer cells occupied: 0
Producer writes   8      Buffer cells occupied: 1
Consumer reads    8      Buffer cells occupied: 0
Producer writes   9      Buffer cells occupied: 1
Consumer reads    9      Buffer cells occupied: 0
Producer writes  10      Buffer cells occupied: 1
Producer done:  Produced values totaling 55
Consumer reads   10      Buffer cells occupied: 0

Consumer done: Read values totaling 55
```

A Note Regarding `BlockingBuffer`'s Output Statements

There's no guarantee that the outputs written by `BlockingBuffer` methods `blockingPut` and `blockingGet` will occur immediately after those methods' corresponding calls to `ArrayBlockingQueue`'s `put` or `take`. So, in some executions, the outputs might appear out of order. Even if the outputs appear out of order, the `ArrayBlockingQueue` object properly synchronizes access to its data, as evidenced by the sums of values written by the producer and read by the consumer.

We Call `Thread` Method `sleep` Only for Demonstration Purposes

We call method `sleep` in the producer and consumer `Runnable`s to emphasize that, **in multithreaded programs, it's unpredictable when each thread will perform its task and for how long it will perform the task when it has a processor**. These thread-scheduling issues are usually beyond the control of Java developers.

We cannot make assumptions about the relative speeds of concurrent threads. Interactions that occur with the operating system, the network, the user and other components can cause the threads to operate at different and ever-changing speeds. When this happens, threads wait. When threads wait excessively, programs become less efficient, interactive programs become less responsive, and applications suffer longer delays.

18.10.7 Bounded Buffers

To minimize the waiting time for threads that share resources and operate at the same average speeds, we can implement a **bounded buffer** that provides a fixed number of buffer cells where the producer can place values and the consumer can retrieve those values. In fact, the `ArrayBlockingQueue` class is a bounded buffer.

With a bounded buffer, if the producer temporarily produces values faster than the consumer can consume them, the producer can write additional values into the extra buffer cells if any are available. This capability enables the producer to perform its task even though the consumer is not ready to retrieve the current value being produced. Similarly, if the consumer temporarily consumes faster than the producer produces new values, the consumer can read additional values (if there are any) from the buffer. This enables the consumer to keep busy even though the producer is not ready to produce additional values. An example of the producer/consumer relationship that uses a bounded buffer is video streaming, which we discussed in Section 18.1.

Even a bounded buffer is inappropriate if the producer and the consumer operate consistently at different speeds. If the consumer always executes faster than the producer, then a buffer containing one location is enough. If the producer always executes faster, only a buffer with an "infinite" number of locations would be able to absorb the extra production. However, if the producer and consumer execute at about the same average speed, a bounded buffer helps to smooth the effects of any occasional speeding up or slowing down in either thread's execution.

The key to using a bounded buffer with a producer and

consumer that operate at about the same speed is providing the buffer with enough locations to handle the anticipated "extra" production. If, over a period of time, we determine that the producer often produces as many as three more values than the consumer can consume, we can provide a buffer of at least three cells to handle the extra production. Making the buffer too small would cause threads to wait longer.

`ArrayBlockingQueue` can work with multiple producers and multiple consumers, ensuring each stored item is read only once. For example, a factory that produces its product very fast will need to have many more delivery trucks (i.e., consumers) to remove those products quickly from the warehousing area (i.e., the bounded buffer) so that the factory can continue to produce products at full capacity.

Generative AI

1 Prompt genAIs, asking how the size of a bounded buffer affects performance and how to determine the optimal size of a bounded buffer in a Java application that implements a producer/consumer relationship.

18.10.8 Other Concurrent Collections

In Chapter 12, we introduced various collections from the Java Collections API. The collections from the `java.util.concurrent` package are specifically designed and optimized for sharing collections among multiple threads. A few of the most commonly used are:

- `ArrayBlockingQueue` (Fig. 18.9)—A fixed-size queue that supports the producer/consumer relationship—possibly with many producers and consumers.
- `ConcurrentHashMap`—A hash-based map (similar to the `HashMap` introduced in Chapter 12) that allows an arbitrary number of reader and writer threads.
- `LinkedBlockingQueue`—A blocking queue implemented as a linked list that can optionally be fixed in size.

The concurrent collections provide their own implementations of various stream-like operations. For example, `ConcurrentHashMap` has methods `forEach`, `reduce` and `search`, which are designed and

optimized for concurrent collections shared among threads. For more information on the concurrent collections, visit

Click here to view code image

```
https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/
   util/concurrent/package-summary.html
```

Generative AI

1 Prompt genAIs, asking when to use `ArrayBlockingQueue` vs. `LinkedBlockingQueue`.

2 Prompt genAIs, asking what determines a `ConcurrentHashMap`'s writer-thread limit.

18.11 Multithreading in JavaFX

JavaFX apps present unique challenges for multithreaded programming. All JavaFX apps have one **JavaFX application thread** that handles interactions with the app's controls, such as rendering them or processing user actions with them. All tasks that require interaction with the GUI are placed in an event queue and executed by the JavaFX application thread in first-in-first-out (FIFO) order.

JavaFX's scene graph is not thread-safe—manipulating its nodes from multiple threads can lead to incorrect results that might corrupt the scene graph. To ensure thread safety in JavaFX you use a technique called **thread confinement** in which you **ensure that programs manipulate the scene graph from only the JavaFX application thread**, not by using thread synchronization techniques. Allowing just one thread at a time to access non-thread-safe objects eliminates the possibility of corruption due to multiple threads accessing these objects concurrently.

It's acceptable to perform brief tasks on the JavaFX application thread—like calculating tips and totals in Chapter 15's Tip Calculator app. However, lengthy tasks performed in the JavaFX application thread prevent it from rendering controls or responding to events until that task completes, making the GUI unresponsive and leading to a poor user experience. So, it's preferable to handle

long-running tasks in separate threads, freeing the JavaFX application thread to continue managing GUI interactions. Of course, you must update the GUI with the computation's results from the JavaFX application thread rather than from the worker thread that performed the calculation.

`Platform` Method `runLater`

JavaFX provides multiple mechanisms for updating the GUI from other threads. One is to call the **`static` method `runLater` of class `Platform`** (package `javafx.application`). This method receives a `Runnable` and schedules it on the JavaFX application thread for execution in the future. Such `Runnable`s should perform only small updates, such as changing the text on a `Label`, so the GUI remains responsive.

Class `Task` and Interface `Worker`

Long-running or compute-intensive tasks should be performed on separate worker threads, not the JavaFX application thread. **Package `javafx.concurrent`** provides interface `Worker` and classes `Task` and `ScheduledService` for this purpose:

- The **`Worker` interface** is for a task that should execute using one or more separate threads.
- The **`Task` class** provides a `Worker` implementation that enables you to perform a task (such as a long-running computation) in a worker thread and update the GUI from the JavaFX application thread based on the task's results. `Task` implements several interfaces, including `Runnable`, so a `Task` object can be scheduled to execute in a separate thread. Class `Task` also provides methods that are guaranteed to update its properties in the JavaFX application thread—this enables programs to bind a `Task`'s properties to GUI controls for automatic updates. Once a `Task` completes, it cannot be restarted, so performing it again requires a new `Task` object. We'll demonstrate `Task`s in the next two examples.
- The **`ScheduledService` class** provides a `Worker` implementation that creates and manages a `Task`. Unlike a `Task`, a `ScheduledService` can be reset and restarted. It can also be configured to automatically restart both after successful completion and if it fails due to an exception.

Generative AI

1 Prompt genAIs for the pros and cons of thread synchronization vs. thread confinement.

18.11.1 Performing Computations in a Worker Thread: Fibonacci Numbers

In the next example, the user enters a number *n*, and the program gets the *n*th Fibonacci number in the **Fibonacci series**,

0, 1, 1, 2, 3, 5, 8, 13, 21, ...

which begins with 0 and 1 and has the property that each subsequent value in the series is the sum of the previous two. In this example, we calculate the *n*th Fibonacci number using a CPU-intensive algorithm that we discuss thoroughly in Section 22.4, so we do not present its details here.

The algorithm is particularly time-consuming for large values of *n*, so we use a `Task` object to perform the calculation in a worker thread. The GUI also provides a separate set of components that displays the next Fibonacci number in the sequence each time the user presses (or selects) a `Button`. This set of components performs its short computation directly in the Java FX application thread. The program can produce up to the 92nd Fibonacci number—subsequent values are outside the range that can be represented by a `long`. Recall that you can use class `BigInteger` to represent arbitrarily large integer values.

Creating a `Task`

Class `FibonacciTask` (Fig. 18.11) extends `Task<Long>` (line 5) to perform the calculation in a worker thread. The instance variable `n` (line 6) represents the Fibonacci number to calculate. Overridden **`Task` method `call`** (lines 14–20) computes the *n*th Fibonacci number and returns the result. The `Task`'s type parameter, `Long` (line 5), determines method `call`'s return type (line 15). Inherited **`Task` method `updateMessage`** (called in lines 16 and 18) updates the `Task`'s `message` property in the JavaFX application thread. In Fig. 18.12, we'll use this to bind a JavaFX control to `FibonacciTask`'s `message` property to display messages while the task is executing.

Click here to view code image

```
1  // Fig. 18.11: FibonacciTask.java
2  // Task subclass for calculating Fibonacci numbers in the background
3  import javafx.concurrent.Task;
4
5  public class FibonacciTask extends Task<Long> {
6     private final int n; // Fibonacci number to calculate
7
8     // constructor
9     public FibonacciTask(int n) {
10       this.n = n;
11    }
12
13    // long-running code to be run in a worker thread
14    @Override
15    protected Long call() {
16       updateMessage("Calculating...");
17       long result = fibonacci(n);
18       updateMessage("Done calculating.");
19       return result;
20    }
21
22    // calculates nth Fibonacci number (Section 22.4 explains this method)
23    public long fibonacci(long number) {
24       if (number == 0 || number == 1) {
25          return number;
26       }
27       else {
28          return fibonacci(number - 1) + fibonacci(number - 2);
29       }
30    }
31 }
```

Fig. 18.11 | `Task` subclass for calculating Fibonacci numbers in the background.

When the worker thread starts running, `FibonacciTask`'s `call` method begins executing. First, line 16 calls the inherited `updateMessage` method to update the `FibonacciTask`'s `message` property, indicating that the task is calculating. Next, line 17 invokes method `fibonacci` (lines 23–30) with instance variable `n`'s value as the argument. When `fibonacci` returns, line 18 updates `FibonacciTask`'s `message` property again to indicate that the calculation completed, then line 19 returns the result to the JavaFX application thread.

`FibonacciNumbers` GUI

The following screen captures demonstrate the app. We labeled the

app's GUI (defined in `FibonacciNumbers.fxml`) with its fx:ids in the third screen capture:

Here, we point out only the key elements and the event-handling methods we'll implement in class `FibonacciNumbersController` (Fig. 18.12). For the complete layout details, open `FibonacciNumbers.fxml` in Scene Builder. The GUI's primary layout is a `VBox` containing two `TitledPane`s. The controller class defines two event handlers:

- `goButtonPressed` is called when the Go `Button` is pressed—this launches the worker thread to calculate a Fibonacci number recursively.
- `nextNumberButtonPressed` is called when the Next Number `Button` is pressed—this calculates the next Fibonacci number in the sequence. Initially, the app displays the Fibonacci of 0 (which is 0).

We do not present the JavaFX `Application` subclass (located in `FibonacciNumbers.java`) because it performs the same tasks we've used previously to load the app's FXML GUI and initialize the controller.

Class `FibonacciNumbersController`

Class `FibonacciNumbersController` (Fig. 18.12)—which we've split into parts for discussion purposes—handles events from two sets of controls:

- The "With FibonacciTask" `TitledPane` provides controls that enable the user to enter a Fibonacci number to calculate and launch a `FibonacciTask` in a worker thread. `Label`s in this `TitledPane` display the `FibonacciTask`'s message property value as it's updated and the final result of the `FibonacciTask` once it's available.
- The "Without FibonacciTask" `TitledPane` provides the Next Number `Button` that enables the user to calculate the next Fibonacci number in sequence. `Label`s in this `TitledPane` display the Fibonacci number being calculated (`"Fibonacci of` *n*`"`) and the corresponding Fibonacci value.

Instance variables `n1` and `n2` (lines 20–21) contain the previous two Fibonacci numbers in the sequence and are initialized to `0` and `1`, respectively. Instance variable `number` (initialized to `1` in line 22)

keeps track of which Fibonacci value will be calculated and displayed next when the user presses (or selects) the Next Number `Button`. The first time this `Button` is pressed (or selected), the Fibonacci of 1 is displayed.

Click here to view code image

```
1  // Fig. 18.12: FibonacciNumbersController.java
2  // Using a Task to perform a long calculation
3  // outside the JavaFX application thread.
4  import java.util.concurrent.Executors;
5
6  import javafx.event.ActionEvent;
7  import javafx.fxml.FXML;
8  import javafx.scene.control.Button;
9  import javafx.scene.control.Label;
10 import javafx.scene.control.TextField;
11
12 public class FibonacciNumbersController {
13    @FXML private TextField numberTextField;
14    @FXML private Button goButton;
15    @FXML private Label messageLabel;
16    @FXML private Label fibonacciLabel;
17    @FXML private Label nthLabel;
18    @FXML private Label nthFibonacciLabel;
19
20    private long n1 = 0; // initialize with Fibonacci of 0
21    private long n2 = 1; // initialize with Fibonacci of 1
22    private int number = 1; // current Fibonacci number to display
23
```

Fig. 18.12 | Using a `Task` to perform a long calculation outside the JavaFX application thread.

Method `goButtonPressed`

When the user presses (or selects) the Go `Button`, method `goButtonPressed` (lines 25–59) executes. Line 29 gets the value entered in the `numberTextField` and attempts to parse it as an integer. If this fails, lines 55–57 prompt the user to enter an integer, select the text in the `numberTextField` and give the `numberTextField` the focus so the user can immediately enter a new value in the `numberTextField`.

Click here to view code image

```
24    // starts FibonacciTask to calculate in background
25    @FXML
```

```
26      void goButtonPressed(ActionEvent event) {
27         // get Fibonacci number to calculate
28         try {
29            int input = Integer.parseInt(numberTextField.getText());
30
31            // create, configure and launch FibonacciTask
32            var task = new FibonacciTask(input);
33
34            // display task's messages in messageLabel
35            messageLabel.textProperty().bind(task.messageProperty());
36
37            // clear fibonacciLabel when task starts
38            task.setOnRunning((succeededEvent) -> {
39               goButton.setDisable(true);
40               fibonacciLabel.setText("");
41            });
42
43            // set fibonacciLabel when task completes successfully
44            task.setOnSucceeded((succeededEvent) -> {
45               fibonacciLabel.setText(task.getValue().toString());
46               goButton.setDisable(false);
47            });
48
49            // launch the FibonacciTask
50            var executor = Executors.newFixedThreadPool(1);
51            executor.execute(task); // start the task
52            executor.shutdown(); // prevent more tasks from being scheduled
53         }
54         catch (NumberFormatException e) {
55            numberTextField.setText("Enter an integer");
56            numberTextField.selectAll();
57            numberTextField.requestFocus();
58         }
59      }
60
```

Line 32 creates a new `FibonacciTask` object, passing the constructor the user-entered value. Line 35 binds the `messageLabel`'s `text` property (a `StringProperty`) to the `FibonacciTask`'s `message` property (a `ReadOnlyStringProperty`). When `FibonacciTask` updates this property, the `messageLabel` displays the new value.

All `Worker`s transition through various states. Class `Task`, an implementation of interface `Worker`, enables you to register listeners for several of these states:

- Lines 38–41 use **`Task` method `setOnRunning`** to register a listener (implemented as a lambda) that's invoked when the

`Task` begins executing—that is, when the `Task` has been assigned a processor and its `call` method is invoked. In this case, we disable the `goButton` so the user cannot launch another `FibonacciTask` until the current one completes. Then, we clear the `fibonacciLabel` (so an old result is not displayed when a new `FibonacciTask` begins).

- Lines 44–47 use **`Task` method `setOnSucceeded`** to register a listener (implemented as a lambda) that's invoked when the `Task` successfully runs to completion. In this case, we call **`Task` method `getValue`** (from interface `Worker`) to get the result, which we convert to a `String` then display in the `fibonacciLabel`. Then, we enable the `goButton` so the user can start a new `FibonacciTask`.

You can also register listeners to be notified when a `Task` is canceled, failed to execute correctly or is scheduled for execution.

Finally, lines 50–52 use an `ExecutorService` to launch the `FibonacciTask`, which schedules it for execution in a separate worker thread. Method `execute` does not wait for the `FibonacciTask` to finish executing. It returns immediately, allowing the GUI to continue processing other events while the computation is performed. Line 52 ensures that no other tasks can be scheduled using the `ExecutorService`. We don't use the `ExecutorSer-vice`'s `AutoCloseable` feature here, because it would block the JavaFX application thread from proceeding as it waits for the task to complete, freezing the GUI.

Method `nextNumberButtonPressed`

If the user presses (or selects) the Next Number `Button`, method `nextNumberButtonPressed` (lines 62–71) executes. Lines 65–66 update the `nthLabel` with the `Fibonacci` number, then update `nthFibonacciLabel` to display `n2`'s value. Next, lines 67–70 add the previous two Fibonacci numbers stored in `n1` and `n2` to determine the next number in the sequence (which will be displayed on the next call to `nextNumberButtonPressed`), update `n1` and `n2` to their new values and increment `number`.

Click here to view code image

```
61      // calculates next Fibonacci value
62      @FXML
63      void nextNumberButtonPressed(ActionEvent event) {
64         // display the next Fibonacci number
```

```
65          nthLabel.setText("Fibonacci of " + number + ": ");
66          nthFibonacciLabel.setText(String.valueOf(n2));
67          long temp = n1 + n2;
68          n1 = n2;
69          n2 = temp;
70          ++number;
71       }
72    }
```

The code for these calculations is in method `nextNumberButtonPressed`, so they're performed on the JavaFX application thread. Handling such short computations in this thread does not cause the GUI to become unresponsive, as with the recursive algorithm for calculating the Fibonacci of a large number. Because the longer Fibonacci computation is performed in a separate worker thread, it's possible to click the Next Number `Button` to get the next Fibonacci number while the recursive computation is still in progress.

18.11.2 Processing Intermediate Results: Sieve of Eratosthenes

Class `Task` provides additional methods and properties that enable you to update a GUI with a `Task`'s intermediate results as the `Task` continues executing in a `Worker` thread. The next example uses the following `Task` methods and properties:

- The **`updateProgress` method** updates a **`Task`'s `progress` property**, which represents the percentage completion.
- The **`updateValue` method** updates a **`Task`'s `value` property**, which holds each intermediate value.

Like `updateMessage`, `updateProgress` and `updateValue` are guaranteed to update the corresponding properties in the JavaFX application thread.

A `Task` to Find Prime Numbers

Figure 18.13 presents class `PrimeCalculatorTask`, which extends `Task<Integer>` to compute the first *n* prime numbers in a worker thread. In the `FindPrimesController` class (Fig. 18.14), we'll bind this `Task`'s `progress` property to a `ProgressBar` control so the app can indicate the `Task`'s completion percentage. The controller also registers a listener for the `value` property's changes—we'll store

each prime value in an `ObservableList` that's bound to a `ListView`.

Click here to view code image

```
// Fig. 18.13: PrimeCalculatorTask.java
// Calculates the first n primes, publishing them as they are found.
import java.util.Arrays;
import javafx.concurrent.Task;

public class PrimeCalculatorTask extends Task<Integer> {
   private final boolean[] primes; // boolean array for finding primes

   // constructor
   public PrimeCalculatorTask(int max) {
      primes = new boolean[max];
      Arrays.fill(primes, true); // initialize all primes elements to true
   }

   // long-running code to be run in a worker thread
   @Override
   protected Integer call() {
      int count = 0; // the number of primes found

      // starting at index 2 (the first prime number), cycle through and
      // set to false elements with indices that are multiples of i
      for (int i = 2; i < primes.length; i++) {
         if (isCancelled()) { // if calculation has been canceled
            updateMessage("Cancelled");
            return 0;
         }
         else {
            try {
               Thread.sleep(10); // slow the thread
            }
            catch (InterruptedException ex) {
               updateMessage("Interrupted");
               return 0;
            }

            updateProgress(i + 1, primes.length);

            if (primes[i]) { // i is prime
               ++count;
               updateMessage(String.format("Found %d primes", count));
               updateValue(i); // intermediate result

               // eliminate multiples of i
               for (int j = i + i; j < primes.length; j += i) {
                  primes[j] = false; // i is not prime
               }
```

```
47                 }
48             }
49         }
50
51         return 0;
52     }
53 }
```

Fig. 18.13 | Calculates the first *n* primes, publishing them as they are found.

Constructor

The constructor (Fig. 18.13, lines 10–13) receives an `int` for the upper limit of the prime numbers to locate, creates the `boolean` array `primes` and initializes its elements to `true`.

Sieve of Eratosthenes

`PrimeCalculatorTask` uses the `primes` array and the **Sieve of Eratosthenes** algorithm to find all primes less than `max`. The Sieve of Eratosthenes takes a list of integers and, beginning with the first prime number, filters out all multiples of that prime. It then moves to the next prime, which will be the next number that's not yet filtered out, and eliminates all of its multiples. It continues until the end of the list is reached and all non-primes have been filtered out. Algorithmically, we begin with element `2` of the `boolean` array and set the cells corresponding to all values that are multiples of 2 to `false` to indicate that they're divisible by 2 and thus not prime. We then move to the next array element, check whether it's `true` and, if so, set all of its multiples to `false` to indicate that they're divisible by the current index. When the whole array has been traversed in this way, all indices that contain `true` are prime, as they have no divisors.

Overridden `Task` Method `call`

`Task` method `call` (lines 16–52) is invoked when the `Task` is executed. The method is `protected` because it's meant to be overridden by a subclass, not called directly. In method `call`, the control variable `i` for the loop (lines 22–49) represents the current index for implementing the Sieve of Eratosthenes. Line 23 calls the inherited **`Task` method `isCancelled`** to determine whether the user has clicked the Cancel button. If so, line 24 updates the `Task`'s `message` property, then line 25 returns `0` to terminate the `Task`

immediately.

If the calculation isn't canceled, line 29 puts the currently executing thread to sleep for 10 milliseconds. We discuss the reason for this shortly. Line 36 calls `Task`'s `update-Progress` method to update the `progress` property in the JavaFX application thread. The percentage completion is determined by dividing the method's first argument by its second argument.

Next, line 38 tests whether the current `primes` element is `true` (and thus prime). If so, line 39 increments the `count` of prime numbers found so far and line 40 updates the `Task`'s `message` property with a `String` containing the `count`. Then, line 41 passes the index `i` to method `updateValue`, which updates `Task`'s `value` property in the JavaFX application thread—in the controller, we'll process this intermediate result and display it in the GUI. When the entire array has been traversed, line 51 returns `0`, which we ignore in the controller (line 53 of Fig. 18.14) because it is not a prime number.

Because the computation progresses quickly, publishing values often, updates can pile up on the JavaFX application thread, causing it to ignore some updates. This is why, for demonstration purposes, we put the worker thread to sleep for 10 milliseconds during each iteration of the loop. The calculation is slowed just enough to allow the JavaFX application thread to keep up with the updates and enable the GUI to remain responsive.

`FindPrimes` GUI

The following screen captures demonstrate the app. We labeled the app's GUI (defined in `FindPrimes.fxml`) with its fx:ids in the third screen capture:

Here, we point out only the key elements and the event-handling methods we'll implement in class `FindPrimesController` (Fig. 18.14). For the complete layout details, open `FindPrimes.fxml` in Scene Builder. The GUI's primary layout is a `BorderPane` containing two `ToolBar`s in the top and bottom areas. The controller class defines two event handlers:

- `getPrimesButtonPressed` is called when the Get Primes `Button` is pressed. This launches the worker thread to find prime numbers less than the value input by the user.
- `cancelButtonPressed` is called when the Cancel `Button` is pressed. This terminates the worker thread.

We do not present the JavaFX `Application` subclass (located in `FindPrimes.java`), because it performs the same tasks we've used previously to load the app's FXML GUI and initialize the controller.

Class `FindPrimesController`

Class `FindPrimesController` (Fig. 18.14)—which we've split into parts for discussion purposes—creates an `ObservableList<Integer>` (lines 24–25) and binds it to the app's `primesListView` (line 31). The controller also provides the event handlers for the Get Primes and Cancel `Button`s.

Click here to view code image

```
1  // Fig. 18.14: FindPrimesController.java
2  // Displaying prime numbers as they're calculated; updating a ProgressBar
3  import java.util.concurrent.Executors;
4
5  import javafx.collections.FXCollections;
6  import javafx.collections.ObservableList;
7  import javafx.event.ActionEvent;
8  import javafx.fxml.FXML;
9  import javafx.scene.control.Button;
10 import javafx.scene.control.Label;
11 import javafx.scene.control.ListView;
12 import javafx.scene.control.ProgressBar;
13 import javafx.scene.control.TextField;
14
15 public class FindPrimesController {
16    @FXML private TextField inputTextField;
17    @FXML private Button getPrimesButton;
18    @FXML private ListView<Integer> primesListView;
19    @FXML private Button cancelButton;
20    @FXML private ProgressBar progressBar;
21    @FXML private Label statusLabel;
22
23    // stores the list of primes received from PrimeCalculatorTask
24    private ObservableList<Integer> primes =
25       FXCollections.observableArrayList();
26    private PrimeCalculatorTask task; // finds prime numbers
27
28    // binds primesListView's items to the ObservableList primes
29    @FXML
30    public void initialize() {
31       primesListView.setItems(primes);
32    }
33
```

Fig. 18.14 | Displaying prime numbers as they're calculated and updating a `ProgressBar`.

Method `getPrimesButtonPressed`

When the user presses (or selects) the Get Primes `Button`, `getPrimesButtonPressed` (lines 35–84) creates a `PrimeCalculatorTask` (line 42) and configures various event listeners:

- Line 45 binds the `statusLabel`'s `text` property to the task's `message` property to update the `statusLabel` automatically as new primes are found.
- Line 48 binds the `progressBar`'s `progress` property to the task's `progress` property to update the `progressBar` automatically with the percentage of completion.
- Lines 51–58 register a `ChangeListener` (using a lambda) that responds to the task's `value` property changes. If the property's `newValue` is not `0` (indicating the task terminated), line 54 adds it to the `primes ObservableList<Integer>`, which is bound to the `primesListView` to display the list's elements. Next, lines 55–56 scroll the `ListView` to its last element so the latest prime number is on the screen.
- Lines 62–72 register listeners for when the task begins running and when the task completes successfully. We use these to enable and disable the app's `Button`s based on the task's state.
- The long-running `PrimeCalculatorTask` sleeps between passing prime values to the JavaFX application thread for display in the `ListView`. Since this task is not CPU-bound, lines 75–77 launch a virtual thread to execute it, potentially enabling better use of your system's CPU resources. **Note that we do not wait for this thread to complete as we have for all of this chapter's virtual thread examples. If the app were to wait here, the event handler would not terminate until the task completes, and the GUI would freeze, preventing other user interactions, such as canceling the task.**

Click here to view code image

```
34      // start calculating primes in the background
```

```
35      @FXML
36      void getPrimesButtonPressed(ActionEvent event) {
37         primes.clear();
38
39         // get Fibonacci number to calculate
40         try {
41            int input = Integer.parseInt(inputTextField.getText());
42            task = new PrimeCalculatorTask(input); // create task
43
44            // display task's messages in statusLabel
45            statusLabel.textProperty().bind(task.messageProperty());
46
47            // update progressBar based on task's progressProperty
48            progressBar.progressProperty().bind(task.progressProperty());
49
50            // store intermediate results in the ObservableList primes
51            task.valueProperty().addListener(
52               (observable, oldValue, newValue) -> {
53                  if (newValue != 0) { // task returns 0 when it terminates
54                     primes.add(newValue);
55                     primesListView.scrollTo(
56                        primesListView.getItems().size());
57                  }
58               });
59
60            // when task begins,
61            // disable getPrimesButton and enable cancelButton
62            task.setOnRunning((succeededEvent) -> {
63               getPrimesButton.setDisable(true);
64               cancelButton.setDisable(false);
65            });
66
67            // when task completes successfully,
68            // enable getPrimesButton and disable cancelButton
69            task.setOnSucceeded((succeededEvent) -> {
70               getPrimesButton.setDisable(false);
71               cancelButton.setDisable(true);
72            });
73
74            // launch the PrimeCalculatorTask
75            var executor = Executors.newVirtualThreadPerTaskExecutor();
76            executor.execute(task); // start the task
77            executor.shutdown();
78         }
79         catch (NumberFormatException e) {
80            inputTextField.setText("Enter an integer");
81            inputTextField.selectAll();
82            inputTextField.requestFocus();
83         }
84      }
```

Method `cancelButtonPressed`

When the user presses the Cancel `Button`, method `cancelButtonPressed` calls the `Prime-CalculatorTask`'s inherited **`cancel` method** (line 90) to terminate the task, then enables the `getPrimesButton` and disables the `cancelButton`.

Click here to view code image

```
86      // cancel task when user presses Cancel Button
87      @FXML
88      void cancelButtonPressed(ActionEvent event) {
89         if (task != null) {
90            task.cancel(); // terminate the task
91            getPrimesButton.setDisable(false);
92            cancelButton.setDisable(true);
93         }
94      }
95   }
```

18.12 Wrap-Up

This chapter presented Java's concurrency capabilities for enhancing performance on multi-core systems. You learned the differences between concurrent and parallel execution. We explained that Java makes concurrency available to you through the language and libraries, and we focused on high-level library features and creating threads to run tasks various ways. You learned that the JVM creates platform threads to run programs and perform housekeeping tasks, such as garbage collection.

We used the `Arrays` class's `sort` and `parallelSort` methods to demonstrate the benefit of a parallel sorting algorithm on a multi-core processor, timing the operations with the Java Date/Time API. You learned that stream pipelines are easy to parallelize, enabling programs to benefit from enhanced performance on multi-core systems. To obtain a parallel stream, you invoked method `parallel` on an existing sequential stream.

We presented the `Runnable` interface for specifying a task that can execute concurrently with other tasks. This interface's `run` method is invoked by the thread executing the task. Then we showed how to use the `Executor` interface to manage the execution

of `Runnable` objects via a thread pool, which can reuse existing platform threads to eliminate the overhead of creating a new thread for each task and can improve performance by optimizing the number of threads to ensure that the processor stays busy.

We explored Project Loom's virtual threads, structured concurrency and scoped values—significant enhancements to Java's concurrency capabilities that address the limitations of traditional platform threads in applications requiring massive numbers of concurrent tasks that spend most of their time blocking. We discussed that virtual threads are lightweight and managed by the JVM rather than the operating system. We compared the initial memory requirements of virtual threads (only a few hundred bytes) and platform threads (one to two megabytes) and presented three ways to create and launch virtual threads. We did detailed performance comparisons, demonstrating how virtual threads significantly outperform platform threads when handling large numbers of concurrent I/O-bound tasks, enabling applications to efficiently manage thousands or millions of concurrent operations. In that example, we discussed the `Callable` and `Future` interfaces, which enable you to execute tasks that return results and obtain those results, respectively.

We discussed structured concurrency and scoped values—preview features at the time of this writing—in the context of a practical weather application example that uses OpenWeatherMap's web services (introduced in Section 11.6). We demonstrated how these features simplify concurrent programming and resource management while making code easier to read, maintain and debug. We used `StructuredTaskScope` to treat related concurrent tasks as a single unit, enabling concurrent programming with straightforward sequential statements. We also used `ScopedValue` objects to share immutable data between tasks launched from the same structured concurrency scope.

You learned that when multiple threads share an object and one or more of them modify that object, indeterminate results may occur unless access to the shared object is managed properly. In an example of the producer/consumer relationship, we showed how to solve this problem via the built-in class `ArrayBlockingQueue`, which handles all the synchronization details for you.

We discussed the fact that JavaFX GUIs are not thread-safe, so all interactions with and modifications to the GUI must be performed in the JavaFX application thread. We also discussed the problems associated with performing long-running calculations in that thread.

Then, we demonstrated how to use class `Task` to perform long-running calculations in worker threads but update the GUI in the JavaFX application thread. You learned how to use a `Task`'s properties to display the results of a `Task` in a GUI when the calculation completes and how to display intermediate results while the calculation is still in process.

In the next chapter, we build API-based Java generative AI applications. You'll use OpenAI APIs to create apps that summarize text, analyze text for sentiment, create accessible descriptions of images, detect a text's language and translate text among languages, generate Java code, perform named entity recognition on text and obtain structured JSON outputs, transcribe speech to text, synthesize speech from text, generate closed captions for a video and more.

Special Section: Generative AI Exercises

For each of the following questions, try multiple genAIs to determine which performs best. Always verify the results. Keep in mind that they sometimes make mistakes and even hallucinate, creating unreliable information.

18.1 ***(Best Practices and Common Pitfalls)*** Prompt genAIs for best practices and common pitfalls of Java multithreaded application development.

18.2 ***(`ThreadLocal` Variables vs. `ScopedValue`s)*** Prompt genAIs, asking for a detailed comparison of the pros and cons of `ThreadLocal` variables vs. `ScopedValue`s with code examples.

18.3 ***(Weather App Enhancement)*** Sometimes, a web service may not respond promptly. Prompt genAIs with the code from Section 18.8's weather app, asking how to make each OpenWeatherMap request stop executing if it has not responded within a specified amount of time (known as "timing out").

18.4 ***(Nested `StructuredTaskScope`s)*** Prompt genAIs for real-world examples of nested `StructuredTaskScope`s and a simple code example demonstrating how to create them.

18.5 ***(Exceptions in `StructuredTaskScope`s)*** Prompt genAIs for a detailed discussion of how exceptions are handled in `StructuredTaskScope`s.

18.6 ***(Custom `StructuredTaskScope`s)*** You can inherit from

`StructuredTaskScope` to customize how their virtual threads are managed. Prompt genAIs to provide code examples demonstrating use-cases for custom `StructuredTaskScope`s.

18.7 ***(Limiting Concurrent Access By Virtual Threads to a Shared Resource)*** Prompt genAIs, asking how to limit the number of concurrent virtual threads accessing a resource.

18.8 ***(Java Flight Recorder)*** Prompt genAIs for an overview of Java Flight Recorder and how it can be used to debug multithreaded applications.

18.9 ***(Fork/Join Framework)*** Prompt genAIs for an overview of Java's Fork/Join framework and how it's used in parallel streams and the `Arrays` methods `parallelSort` and `parallelSetAll`.

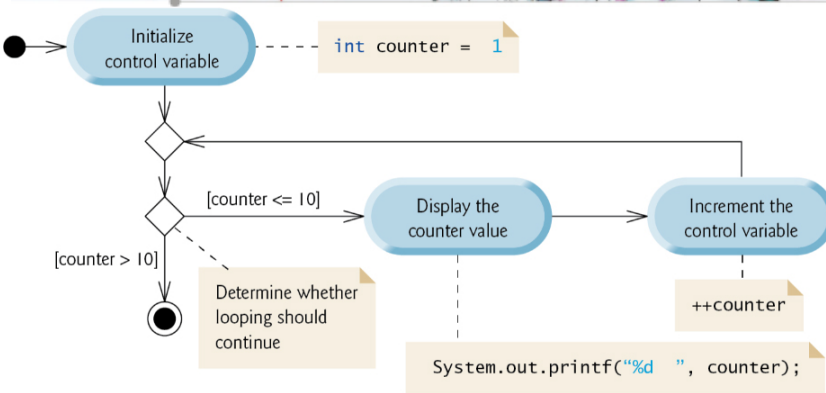
18.10 ***(Uncaught Exceptions in Multithreaded Applications)*** Prompt genAIs, asking them to list cases where an uncaught exception does not terminate a Java program.

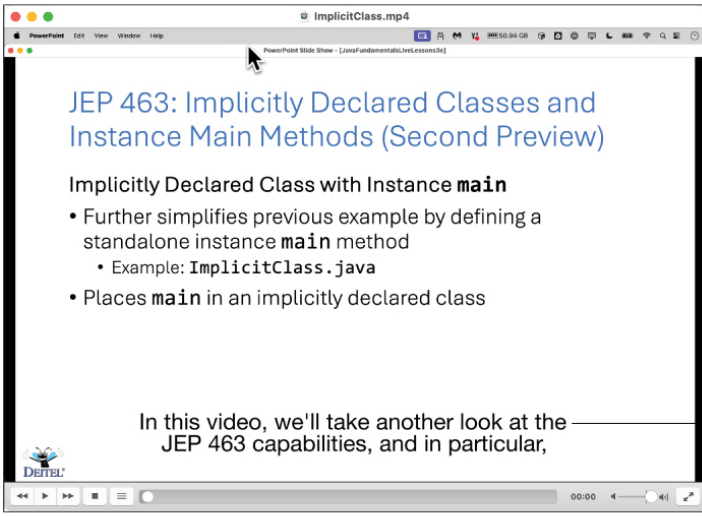
18.11 ***(Deadlock, Livelock and Indefinite Postponement)*** Three problems that can occur in systems that allow threads to wait (such as the producer/consumer relationship) are

- deadlock—in which one or more threads will wait forever for an event that cannot occur,
- livelock—in which threads remain active but keep retrying or adjusting in a way that prevents completion of their tasks and
- indefinite postponement—in which one or more threads will be delayed for some unpredictably long time.

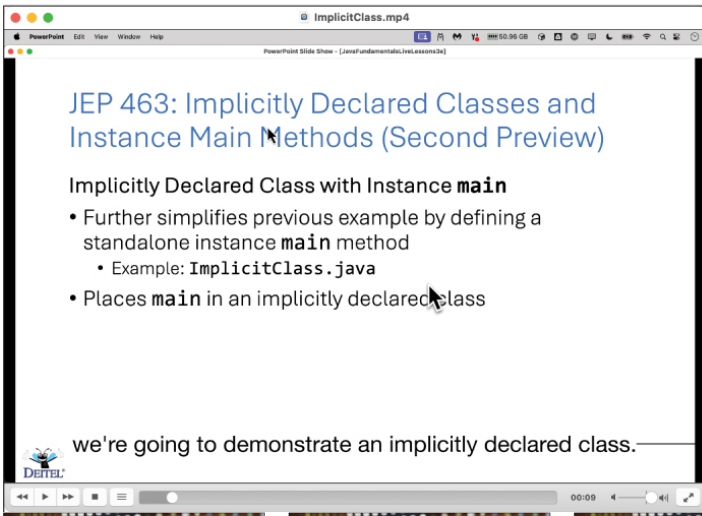
Prompt genAIs, asking them to discuss how to avoid deadlock, livelock and indefinite postponement in multithreaded Java programs.

18.12 ***(Bounded Buffer: A Real-World Example)*** Prompt genAIs to describe how a highway off-ramp onto a local road is a good example of a producer/consumer relationship with a bounded buffer and how the designers might choose the size of the off-ramp.

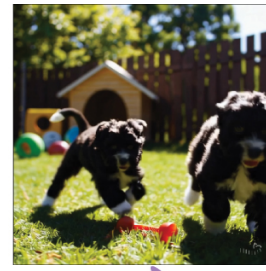

Initialize control variable
int counter = 1
[counter <= 10]
Display the counter value
Increment the control variable
[counter > 10]
Determine whether looping should continue
++counter
System.out.printf("%d ", counter);

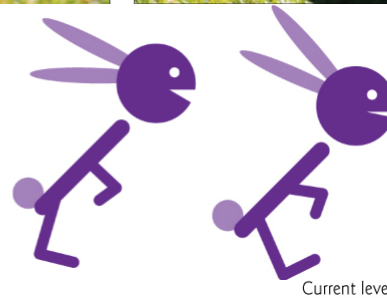
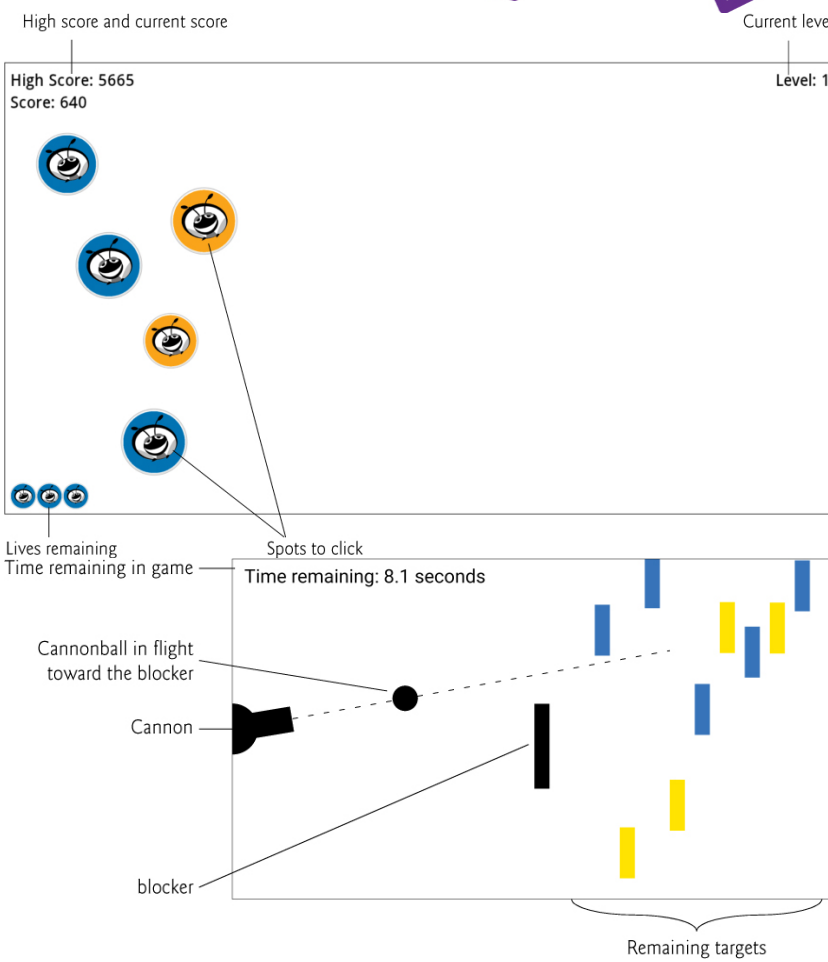


generated closed caption



generated closed caption





High score and current score
Current level
High Score: 5665
Score: 640
Level: 1
Lives remaining
Spots to click
Time remaining in game
Time remaining: 8.1 seconds
Cannonball in flight toward the blocker
Cannon
blocker
Remaining targets

Building API-Based Java Generative AI Applications

Objectives

In this chapter, you'll:

- Build Java applications based on the OpenAI APIs using the open-source Simple-OpenAI Java library.
- Build multi-modal AI apps.
- Generate text.
- Summarize text.
- Extract key points from text.
- Analyze the sentiment of text.
- Get detailed, accessible descriptions of images by recognizing objects in images.
- Translate text among natural languages.
- Generate Java code.
- Perform named-entity recognition (NER) on a body of text.
- Receive and process structured outputs.
- Transcribe speech to text.
- Synthesize speech from text.
- Generate images.
- Generate video with OpenAI's Sora.
- Create closed captions for a video from its audio track.
- Generate brief video segments.

Outline

19.1 Introduction

19.2 OpenAI APIs

19.2.1 Some of OpenAI's Models

19.2.2 OpenAI API Capabilities

19.2.3 Get an OpenAI Developer Account

19.2.4 OpenAI Developer API Key

19.3 Setting Up a Java Environment

19.3.1 Simple-OpenAI Open Source Library: Connecting to OpenAI from Java

19.3.2 IntelliJ Maven-Based Java Project

19.4 Text Generation Via Chat Completions

19.4.1 Text Summarization

19.4.2 Sentiment Analysis

19.4.3 Accessible Image Descriptions

19.4.4 Language Detection and Translation

19.4.5 Code Generation

19.4.6 Named Entity Recognition (NER) and Structured Outputs

19.5 Speech Synthesis and Speech Recognition

19.5.1 English Speech-to-Text for Audio Transcription

19.5.2 Text-to-Speech

19.6 Image Generation

19.7 Video

19.7.1 Generating Video Closed Captions

19.7.2 Sora

19.7.3 Other Video GenAIs

19.8 Moderation

19.9 Class `OpenAIUtilities`

19.9.1 Builder Design Pattern

19.9.2 `package` and `import` Statements

19.9.3 Nested `record` Class `Message`
19.9.4 `SimpleOpenAI` Client Object
19.9.5 `chat` Method
19.9.6 Helper Method `makeChatMessages`
19.9.7 `describeImage` Method
19.9.8 `translate` Method
19.9.9 `chatWithStructuredOutput` Method
19.9.10 `speechToText` Method
19.9.11 `textToSpeech` Method
19.9.12 `image` Method
19.9.13 `speechToVTT` Method
19.9.14 `checkPrompt` Method
19.10 Wrap-Up
GenAI Prompt Exercises

Project Categories:

Text-Based Apps; Creative-Writing Apps; Text-to-Speech and Speech-to-Text Apps; Spoken-Language Translation Apps; Code-Based Apps; Image-Based Apps; Audio, Speech and Music Apps; Video; Multimodal Apps; Music-Based Apps; Education-Based Apps; Game Development; Assistant-Based Apps; Special Feature: Crossword Puzzles; Capstone Project: The Multimodal GenAI Debating Society

19.1 Introduction

So far, you've manually prompted genAIs via their web interfaces and compared and used their results. We did not show these prompts' results. In this chapter, we program Java API-based genAI applications and show their outputs or play the generated audios and videos.

Multimodal GenAIs

When genAI chatbots were introduced, they received a text prompt

and generated text or an image. Many are now **multimodal**—they can understand text, code, images, audio and video inputs and use them to generate mixed text, code, images, audio and video outputs.[1,2,3] For example, you can prompt a genAI with an audio file containing speech and instructions, asking it to transcribe the speech into text, display the transcription, analyze its sentiment, translate it to another spoken language and synthesize speech in that other language.

1. "What Is Multimodal AI?" DataCamp. Accessed February 25, 2025. https://www.datacamp.com/blog/what-is-multimodal-ai.
2. "What Is Multimodal AI?" IBM. Accessed February 25, 2025. https://www.ibm.com/think/topics/multimodal-ai.
3. "Generative Artificial Intelligence." Wikipedia. Wikimedia Foundation. Accessed February 25, 2025. https://en.wikipedia.org/wiki/Generative_artificial_intelligence.

Various genAIs allow you to speak to them and receiving spoken responses. In fact, at OpenAI DevDay 2024, OpenAI CEO Sam Altman said that ChatGPT's Advanced Voice Mode was so realistic that it made him think he was chatting with another person.[4]

4. Zachary Boddy, "Sam Altman Says ChatGPT's Voice Mode Was the First Time He Was Tricked into Thinking an AI Was a Person," Windows Central, February 15, 2024. Accessed February 25, 2025. https://www.windowscentral.com/software-apps/sam-altman-says-chatgpts-voice-mode-was-the-first-time-he-was-tricked-into-thinking-an-ai-was-a-person.

OpenAI APIs

We'll use the OpenAI APIs to build multimodal Java genAI applications. Companies like Google, Anthropic, Meta, Hugging Face and more also provide APIs.[5] The OpenAI API does not have an official Java library, so we'll use the free, open-source Simple-OpenAI library (discussed in Section 19.3) to interact with the OpenAI APIs and process their results. As you'll see, prompts like those you've used throughout this book are essential to genAI API-based programming.

5. It's expensive for companies like OpenAI, Google and Anthropic to train the models we access here via APIs. So, their web-based APIs are typically paid services. Some offer limited free tiers.

Generative AI APIs with Free Tiers

Google Gemini and Hugging Face have free tiers. For an overview these, visit:

- Google Gemini pricing tiers: https://ai.google.dev/pricing
- Hugging Face pricing tiers: https://huggingface.co/pricing

Hugging Face is an open-source platform specializing in natural language processing, providing pretrained language models (almost 1.5 million of them), datasets and tools for working with models. Meta offers its Llama 3 genAI models free for download to run locally. For more information, visit https://www.llama.com/. Meta's Llama models are also available on Hugging Face.

Caution on Using Paid GenAI APIs

Generative AI services often charge based on usage, which can result in significant fees depending on request volume and task complexity. Review the pricing structures of paid APIs and monitor your usage to avoid unexpected charges. Free tiers typically have strict usage limits, so exceeding these can lead to substantial bills. Always check for available budgeting tools and cost management strategies the API services provide.[6]

6. *ChatGPT* response to "Suggest a warning I can provide about paid generative AI APIs and that readers should be careful not to run up large bills." October 25, 2024. https://chatgpt.com.

Always be cautious when using paid services. In your OpenAI developer account, you can manage your billing at:

Click here to view code image

```
https://platform.openai.com/settings/organization/billing/overview
```

Setting "auto recharge" to "off" prevents accidental charges.

End-of-Chapter Projects

This chapter includes an enormous number of end-of-chapter projects. We "let our minds run wild" with fun and interesting possibilities. Some projects can be solved using just the OpenAI APIs we present in this chapter. Many will require additional research and Simple-OpenAI library features we do not cover.

Generative AI

1 Prompt genAIs for a list of generative AI APIs that are free or offer free tiers.

19.2 OpenAI APIs

OpenAI (https://openai.com/) is one of many genAI providers and the creator of

- the ChatGPT web-based multimodal AI chatbot at https://chatgpt.com,
- the Dall-E AI image generator, which is now integrated into ChatGPT's web interface (though still separate in the OpenAI APIs) and
- the Sora[7] AI video generator at https://sora.com.

7. "Creating video from text." Accessed February 25, 2025. https://openai.com/index/sora/.

OpenAI's APIs provide access to the AI models behind ChatGPT, Dall-E and more. There were no plans for a Sora video-generation API at the time of this writing.[8]

8. "OpenAI Says It Has No Plans for Sora," Yahoo Finance. Accessed February 25, 2025. https://finance.yahoo.com/news/openai-says-no-plans-sora-203837499.html.

19.2.1 Some of OpenAI's Models

Here are some OpenAI models you can access via the OpenAI APIs. These will change over time. For the latest complete list, visit https://platform.openai.com/docs/models:

- **Reasoning models**—Models for tasks that require complex reasoning. The o3-mini model is "particularly good at coding, math, and science."[9]

  9. "Models—o1 and o1-mini." OpenAI Documentation. Accessed February 25, 2025. https://platform.openai.com/docs/models/gp#o1.

- **GPT-4o**—High-intelligence model that can handle complex, multi-step tasks.
- **GPT-4o mini**—Smaller, more affordable model than GPT-4o meant for fast, lightweight tasks.
- **GPT-4o and GPT-4o-mini Realtime**—Models that process realtime audio and text inputs and outputs.
- **GPT-4o and GPT-4o-mini Audio**—Models that generate a spoken audio summary of text (text in, audio out), perform

sentiment analysis on spoken audio (audio in, text out) and handle asynchronous speech-to-speech interactions with a model (audio in, audio out).

- **DALL-E**—Generates/edits images in response to natural language prompts.
- **TTS**—Converts text to speech.
- **Whisper**—Transcribes and translates audio.

19.2.2 OpenAI API Capabilities

The OpenAI API's capabilities are divided into categories, including:

- **Text generation** (Section 19.4)[10]—generate text based on text prompts.

  10. "Generate text from a prompt." Accessed February 25, 2025. https://platform.openai.com/docs/guides/text-generation/generate-text-from-a-prompt.

- **Image generation** (Section 19.6)[11]—generate images based on text prompts, edit existing images and provide variations on an existing image.

  11. "Image generation." Accessed February 25, 2025. https://platform.openai.com/docs/guides/images/image-generation.

- **Vision** (Section 19.4.3)[12]—answer questions about image inputs.

  12. "Vision." Accessed February 25, 2025. https://platform.openai.com/docs/guides/vision.

- **Audio generation**[13]—receive a text input and generate a spoken audio summary of the text, perform sentiment analysis on spoken audio and enable speech-to-speech model interactions.

  13. "Audio generation." Accessed February 25, 2025. "Vision." Accessed February 25, 2025. https://platform.openai.com/docs/guides/audio.

- **Text to speech** (Section 19.5)[14]—narrate text, synthesize speech in 57 languages and stream real-time audio output.

  14. "Text to speech." Accessed February 25, 2025. "Vision." Accessed February 25, 2025. https://platform.openai.com/docs/guides/text-to-speech/.

- **Speech to text** (Section 19.5)[15]—transcribe speech in 57

languages into text in the spoken language and transcribe non-English speech directly to English text.

15. "Speech to text." Accessed February 25, 2025. https://platform.openai.com/docs/guides/speech-to-text.

- **Vector embedding**16—convert text to numerical values that are used to determine how text strings relate to one another.

16. "Embeddings." Accessed February 25, 2025. https://platform.openai.com/docs/guides/embeddings.

- **Moderation** (Section 19.8)17—determine whether images or text are harmful.

17. "Moderation." Accessed February 25, 2025. https://platform.openai.com/docs/guides/moderation.

- **Reasoning**18—perform complex reasoning for computer programming and solving math, physics, biology and chemistry problems.

18. "Reasoning models." Accessed February 25, 2025. https://platform.openai.com/docs/guides/reasoning/reasoning.

19.2.3 Get an OpenAI Developer Account

The OpenAI examples require an OpenAI developer account. OpenAI's APIs are fee-based services. Follow the instructions at the following site to sign up for an account:

Click here to view code image

```
https://platform.openai.com/signup
```

To execute this chapter's examples live, you must add money to your account.19 The examples we present do not cost a lot to execute.20 You can learn about OpenAI's pricing for various models at https://openai.com/api/pricing.

19. For readers who do not want to sign up for paid services, we try to provide enough information in the text so you will understand how these capabilities work. We expect the range and quality of available free services to improve over time.

20. On February 14, 2024, we deposited $50 into our OpenAI developer account. As of February 28, 2025, we'd spent only $16 running this chapter's examples frequently for testing and development, and demonstrating the APIs in Paul Deitel's Python Full Throttle live trainings on O'Reilly Online Learning.

19.2.4 OpenAI Developer API Key

You need a developer key to invoke OpenAI's APIs. Perform the following steps to get one:

- Sign into your account at `https://platform.openai.com/docs/overview`.
- Press (or select) the Start building button in the page's upper-right corner—after you create your first API key below, this button will change to a settings icon ( ).
- Fill in the Organization name and What best describes you? fields then press (or select) the Create organization button.
- In the Invite your team step, you can add email addresses for other team members. Press (or select) Continue.
- In the Make your first API call step, specify an API key name and a Project name, then press (or select) the Generate API Key button. (If you create additional API keys in the future, you'll select API Keys the left column of your account's settings page.)
- Press or select the Copy button to copy the alphanumeric key to the clipboard.

For security, you'll store your API key in an environment variable on your system rather than embedding it in your code, as discussed below.

Storing the API Key in macOS

Perform the following steps to store your API key on macOS:

- Open Terminal.
- Open `~/.zshrc` using a text editor (e.g., `nano ~/.zshrc`).
- Scroll to the end of the file and add the following line, replacing *YourAPIKey* with the lengthy alphanumeric API key you copied previously.

  Click here to view code image

  ```
  export OPENAI_API_KEY="YourAPIKey"
  ```

- Save and close the file.

- Run the following command in the Terminal:

```
source ~/.zshrc
```

Storing the API Key on Windows

Perform the following steps to store your API key on Windows:

- In the taskbar's Search field, enter `SystemPropertiesAdvanced`, and press or select *Enter*.
- In the System Properties dialog, press or select the Environment Variables… button.
- Under User variables, click New….
- Enter the Variable name as `OPENAI_API_KEY` and, for the Variable value, paste in the lengthy alphanumeric API key you copied previously.
- Click OK to save the environment variable, then click OK in the System Properties dialog to close it.

19.3 Setting Up a Java Environment

OpenAI's APIs are implemented as web services. We'll use the following Java library that encapsulates the web-service interactions, making it convenient to use OpenAI's APIs.

19.3.1 Simple-OpenAI Open Source Library: Connecting to OpenAI from Java

OpenAI does not provide an official Java library, but several third-party options are available. In this chapter, we use Simple-OpenAI—"a Java library to use the OpenAI API in the simplest possible way."[21] This library supports most major OpenAI API capabilities, including chat completions, speech synthesis, speech recognition, image generation, image editing, structured outputs, realtime speech-to-speech, and more.[22] The library provides classes and methods that encapsulate the details of

21. Estela, Sashir. "Simple-OpenAI." Accessed February 25, 2025. https://github.com/sashirestela/simple-openai.

22. Estela, Sashir. Simple-OpenAI, Accessed February 25, 2025, https://github.com/sashirestela/simple-openai.

- authenticating with OpenAI using your `OPENAI_API_KEY` so you can use their web services,
- managing files used in your requests,
- formulating web-service requests that invoke OpenAI's APIs,
- handling the responses they return (such as generated text, images and audio), and more.

For a complete list of the OpenAI services Simple-OpenAI supports, visit the following site and select Supported Services in the Table of Contents:

Click here to view code image

```
https://github.com/sashirestela/simple-openai
```

19.3.2 IntelliJ Maven-Based Java Project

This chapter's examples each call methods of our custom class `OpenAIUtilities` (discussed in Section 19.9) from package `deitel.openai`, which uses the Simple-OpenAI Java library to interact with the OpenAI APIs. This enables us to focus on each example's genAI concepts and present the Simple-OpenAI details later. Class `OpenAIUtilities` also allows us to share code among our applications. When compiling and running this chapter's apps, you'll use our custom library file `OpenAIUtilities.jar`, which is located in the `libs` subfolder with this chapter's examples.

To reduce the complexity of compiling our class `OpenAIUtilities` (for example, if you add to it while solving this chapter's exercises), it's defined in an IntelliJ IDEA Community Edition IDE project containing the class's code and its dependencies:

- the **Jackson library for JSON deserialization**,
- the **Simple-OpenAI library for interacting with the OpenAI APIs** and
- the **Simple-OpenAI library's various dependencies**.

In our IntelliJ IDEA Community Edition test-drive at

```
https://deitel.com/javafp5
```

we discuss how to set up basic projects for running most of this book's examples, and how to set up an Apache Maven[23] project to

automatically manage library dependencies. The test-drive also discusses compiling our `OpenAIUtilities` class's project and creating the `OpenAIUtilities.jar` file, which you might need when solving the exercises.

23. "Apache Maven Project." Accessed February 25, 2025. https://maven.apache.org/.

19.4 Text Generation Via Chat Completions

In this section, we present fully coded apps that interact with OpenAI's `"gpt-4o"` model to request chat completions:

- Section 19.4.1 performs **text summarization**—in two OpenAI API requests, we upload a Deitel video transcript and ask the chat API to return a summary abstract paragraph, followed by a list of the transcript's key points. Text responses like these are known as **chat completions**.
- Section 19.4.2 uses a chat-completion request to perform **sentiment analysis**—determining whether the Deitel video transcript is positive, neutral or negative. We also ask the genAI to explain how it came to its conclusion.
- Section 19.4.3 asks OpenAI to **describe images for accessibility**. This multi-modal application uploads an image and requests a detailed image description. We get detailed descriptions of Section 4.3's `for`-loop UML diagram and a scenic Aruba beach photo.
- Section 19.4.4 uses chat-completion requests to **translate text**. OpenAI will auto-detect the text's source language then translate the text to a specified target language. We translate text from English to Spanish and Japanese then translate the Spanish and Japanese text back to English.
- Section 19.4.5 uses a chat-completion request to **generate Java code** for a simulation that summarizes 600,000,000 six-sided die rolls and displays the results. We then compile and execute the generated code.
- Section 19.4.6 uses a chat-completion request to perform the natural language processing (NLP) task **named-entity recognition (NER)**, which attempts to locate and categorize text mentions like dates, times, quantities, places, people,

things, organizations, etc. This example demonstrates OpenAI's ability to produce **structured output** in your application's required format. To make the results easier to process, we provide the API with the exact JSON response format we wish to receive, then use the Jackson open-source library (Chapter 11) to process the result. NER is one of many natural language processing tasks OpenAI's APIs can perform. Section 7.7 listed many common NLP tasks you could try.

19.4.1 Text Summarization

Text summarization is a natural language processing (NLP) task that analyzes a body of text and produces a shorter version containing the key ideas. This is useful for getting quick summaries of items like articles, academic papers, book chapters or sections, lengthy emails, reports, study materials, product reviews and more. Figure 19.1 summarizes a presentation transcript as

- a summary abstract paragraph and
- a key-points list.

The file `transcript.txt` in the `ch19` examples folder's `resources` subfolder contains the original presentation transcript. Many of this chapter's examples use files from this folder, and those that generate audio, images and Java code write their results into files in the `resources` folder's `outputs` subfolder. Throughout this chapter's examples, we assume the book's code is located in your user account's `Documents` folder in the `examples` subfolder, as discussed in the Before You Begin section. **If you placed them elsewhere, you must update the files paths we specify in the examples.**

Compiling and Running this Chapter's Examples

To compile each of this chapter's examples on macOS, open a Terminal window, change to the folder containing the example, then enter:

Click here to view code image

```
javac -cp .:../libs/OpenAIUtilities.jar SourceFile
```

where *SourceFile* is the name of the Java source-code file for the example—in this section, `TextSummarizationDemo.java`. To run the

code, enter the following command

Click here to view code image

```
java -cp .:../libs/OpenAIUtilities.jar ClassName
```

On Windows, open a PowerShell or Command Prompt window, change to the folder containing the example, then execute the preceding commands, replacing the forward slashes (`/`) with backslashes (`\`) and the colons (`:`) with semicolons (`;`).

Class `TextSummarizationDemo`

We split the `TextSummarizationDemo` class (Fig. 19.1) into parts for discussion purposes. Line 3 imports our class `OpenAIUtilities`, and line 4 imports its nested class `Message` (discussed momentarily).[24]

24. The ideas for some of this chapter's demos were inspired by OpenAI's tutorial, "Creating an automated meeting minutes generator with Whisper and GPT-4." Accessed February 25, 2025. https://platform.openai.com/docs/tutorials/meeting-minutes. The prompts and Java code we present throughout the chapter are original.

Click here to view code image

```
1  // Fig. 19.1: TextSummarizationDemo.java
2  // Summarizing a transcript as an abstract paragraph and key points.
3  import deitel.openai.OpenAIUtilities;
4  import deitel.openai.OpenAIUtilities.Message;
5
6  import java.nio.file.Files;
7  import java.nio.file.Path;
8  import java.util.List;
9
10 public class TextSummarizationDemo {
11    public static void main(String[] args) throws Exception {
```

Fig. 19.1 | Summarizing a transcript as an abstract paragraph and key points.

Loading the Presentation Transcript

Lines 13–14 use the `Path.of` method (introduced in Section 11.3) to create a `Path` to the file `transcript.txt`. Line 15 uses the **`Files` `static` method `readString`** to read the entire file into a `String`.

Click here to view code image

```
// load transcript.txt
Path transcriptPath = Path.of(System.getProperty("user.home"),
   "Documents", "examples", "ch19", "resources", "transcript.txt");
String transcript = Files.readString(transcriptPath);

```

Requesting a Chat Completion That Produces a Summary Abstract Paragraph

Lines 19–31 call our `OpenAIUtilities` class's `static chat` method (discussed in Section 19.9.5) to request an OpenAI chat completion that produces a summary abstract paragraph for the transcript loaded in line 15. Our `chat` method uses the Simple-OpenAI library to interact with OpenAI APIs that respond to prompts, similar to when you interact with ChatGPT through its web interface. **When you run this and subsequent examples, keep in mind that the responses returned by genAIs typically change from request to request, even if your prompt is identical.**

Click here to view code image

```
// get a summary abstract with OpenAI's gpt-4o model
System.out.println("CREATE A SUMMARY ABSTRACT OF A TRANSCRIPT");
String summaryAbstract = OpenAIUtilities.chat("gpt-4o",
   List.of(
      new Message("system", """
         Given a transcript of a technical presentation, create a
         concise, clear summary abstract in paragraph form, written
         in a direct style that avoids prepositional phrases and
         uses straightforward sentence structures. Focus on the key
         points without referring to the speaker. Capture the key
         ideas, so a person can understand it without reading the
         full transcript."""),
      new Message("user", transcript)
   )
);
System.out.printf("%s%n%n", summaryAbstract);

```

```
CREATE A SUMMARY ABSTRACT OF A TRANSCRIPT
The "What's New in Java" lesson introduces updates and features from Java 10
to Java 22, with a brief mention of a Java 23 feature. These new elements are
integrated into various editions, including Java Fundamentals Third Edition
and others. The course is aligned with existing lessons to highlight how
these features can be applied, although placements in new products may vary.
The lesson covers core areas such as Java fundamentals, control statements,
data structures, and object-oriented programming. Key topics include func-
```

```
tional programming with lambdas and streams, JavaFX for graphical interfac-
es, and advanced concepts like concurrency and generative AI. The lesson also
explores practical tools like JShell for interactive development. Java ver-
sion-specific features are highlighted for developers operating on different
Java platforms, with an emphasis on long-term support releases. The new con-
tent will gradually replace older lessons as it is incorporated into updates.
```

Messages Passed to the OpenAI APIs

Many OpenAI API calls require messages consisting of a `role` and `content`. The role is `"user"`, `"system"` or `"assistant"`—discussed momentarily. The `content` is typically

- a prompt like you'd input via a genAI's web interface or
- data the API call will process.

In our code, we defined these messages using our `OpenAIUtilities` class's nested `record` class `Message` (imported at line 4 and discussed in Section 19.9.3):

Click here to view code image

```
public record Message(String role, String content) {}
```

Several of our `OpenAIUtilities` class's methods, including `chat`, receive `List<Message>` objects. Our `chat` method's first argument specifies the OpenAI model to use to formulate a response—the **`"gpt-4o"` model** was one of OpenAI's most up-to-date and powerful models at the time of this writing. Each item in the `List<Message>` passed as `chat`'s second argument is a `Message` object containing a `role` and `content`.

`"system"` Role

A message with the **`"system"` role**[25] prompts the model with details to help it perform a task. The corresponding `content` typically explains what you want the model to do.[26] For example, lines 21–28 define a text block describing how we'd like the model to summarize text. If you were asking the model to write Java code, you might tell it that it's an expert in Java programming. Instructions like this give the model context to help it provide more relevant results.

25. "Messages and Roles." Accessed February 25, 2025. https://platform.openai.com/docs/guides/text-generation/#messages-and-roles.

26. OpenAI recently added `"developer"` messages that are specifically for use with OpenAI's "reasoning models" (such as `"o1"` or `"o3-mini"`). According to https://community.openai.com/t/how-is-developer-message-better-than-system-prompt/1062784/2, if you submit a `"developer"` message to the `"gpt-4o"` model (or another GPT model), it's converted to a `"system"` message, and if you submit a `"system"` message to reasoning model, it's converted to a `"developer"` message.

`"user"` Role

A message with the **`"user"` role**[27] provides as its `content` a prompt for a task to perform or data the model requires to perform a task. You can supply multiple `"system"` and `"user"` messages —`"system"` messages are prioritized over `"user"` messages. Line 29 provides the data to summarize (`transcript`). If you'd like the model to generate code, the `content` would indicate what you'd like the code to do.

27. "Messages and Roles." Accessed February 25, 2025. https://platform.openai.com/docs/guides/text-generation/#messages-and-roles.

`"assistant"` Role

A message with the **`"assistant"` role**[28] is typically used to provide examples to the model, such as showing it the exact format you want for a response. Unlike the ChatGPT web interface, the chat-completion API does not have memory, so it does not remember previous chat-completion requests. So, messages with the `"assistant"` role can also be used to supply a previous chat-completion response in a conversation your code is having with the genAI, giving it more context for the current request. For more details on prompt engineering and these roles, visit

28. "Messages and Roles." Accessed February 25, 2025. https://platform.openai.com/docs/guides/text-generation/#messages-and-roles.

Click here to view code image

```
https://platform.openai.com/docs/guides/prompt-engineering
```

Extracting Key Points

To extract key points from the transcript, we use class `OpenAIUtilities`' `chat` method but lines 36–45 change the `"system"` prompt to ask OpenAI to generate a list of the top five key points from the original audio transcription. In our experience, the

resulting key points are sometimes returned as a numbered list and sometimes as a bullet list, so we explicitly asked for a numbered list in this prompt.

Click here to view code image

```
        // get key points with OpenAI's gpt-4o model
        System.out.println("EXTRACT KEY POINTS FROM A TRANSCRIPT");
        String keyPoints = OpenAIUtilities.chat("gpt-4o",
           List.of(
              new Message("system", """
                 Given a transcript of a technical presentation, identify
                 the top 5 key points and present them as a numbered list.
                 For each point, use a concise, direct, clear, and
                 straightforward sentence. Avoid prepositional phrases."""),
              new Message("user", transcript)
           )
        );
        System.out.printf("%s%n", keyPoints);
     }
  }
```

```
EXTRACT KEY POINTS FROM A TRANSCRIPT
1. The lesson covers new Java programming features from versions Java 10 to
Java 22 with a preview of Java 23.
2. Java's recent features include executing source files directly and simpli-
fied main methods for easier coding.
3. Java 14 introduced switch expressions with powerful pattern matching capa-
bilities.
4. Java 21's sequenced collections offer unified method naming and reverse
iteration for ease of use.
5. Sealed classes and interfaces in Java give hierarchy designers complete
control over subclassing and implementation.
```

Generative AI

1 Prompt genAIs, asking why you'd provide multiple `"system"` or `"user"` messages for an OpenAI API chat completion.

2 Prompt genAIs, asking for examples of using `"assistant"` messages for an OpenAI API chat completion.

19.4.2 Sentiment Analysis

Figure 19.2 analyzes the sentiment of the text in `transcript.txt`.

This example uses our `OpenAIUtilities` class's `chat` method with the `"gpt-4o"` model again, but lines 21–25 change the `"system"` prompt to ask for a chat completion that analyzes the sentiment of the original transcript and explains how the model came to its conclusion. The model's response indicates that the transcript's sentiment is neutral. It also broke its conclusion into segments, indicating:

- the content is "Objective and Informative,"
- the speaker has a "Lack of Personal Emotional Expression,"
- the speaker maintains a "Balanced Tone" and
- the speaker provides "Constructive and Supportive Notes,"

each with explanatory notes.

Click here to view code image

```
1  // Fig. 19.2: SentimentAnalysisDemo.java
2  // Analyzing the sentiment of a transcript.
3  import deitel.openai.OpenAIUtilities;
4  import deitel.openai.OpenAIUtilities.Message;
5
6  import java.nio.file.Files;
7  import java.nio.file.Path;
8  import java.util.List;
9
10 public class SentimentAnalysisDemo {
11    public static void main(String[] args) throws Exception {
12       // load transcript.txt
13       Path transcriptPath = Path.of(System.getProperty("user.home"),
14          "Documents", "examples", "ch19", "resources", "transcript.txt");
15       String transcript = Files.readString(transcriptPath);
16
17       // analyze text sentiment with OpenAI's gpt-4o model
18       System.out.println("ANALYZE SENTIMENT");
19       String sentiment = OpenAIUtilities.chat("gpt-4o",
20          List.of(
21             new Message("system", """
22                You are an expert in sentiment analysis. Analyze the
23                following presentation transcript and state whether
24                the sentiment is positive, negative, or neutral.
25                Explain your analysis."""),
26             new Message("user", transcript)
27          )
28       );
29       System.out.printf("%s%n%n", sentiment);
30    }
```

```
31     }

ANALYZE SENTIMENT
The sentiment of the presentation transcript is **neutral**.

### Analysis:

1. **Objective and Informative**:
   - The content is highly objective and focused on providing information
about the new features and updates in the upcoming Java course.
   - There is a detailed breakdown of what will be covered in the lesson, in-
cluding the structure and individual topics from fundamentals to high-end
Java concepts.

2. **Lack of Personal Emotional Expression**:
   - The speaker does not express personal emotions or opinions that would
indicate a positive or negative sentiment.
   - The language is professional and focused on conveying the course de-
tails.

3. **Balanced Tone**:
   - The transcript maintains a balanced tone throughout, avoiding any emo-
tionally charged language.
   - There are no exclamatory remarks or emotionally connotative words which
could sway the sentiment in either a positive or negative direction.

4. **Constructive and Supportive Notes**:
   - The speaker provides guidance on how to reach out for any questions and
mentions the availability of updates and support, which is informative but
does not express a particularly strong emotional sentiment.

Conclusively, the presentation's purpose is to inform the audience about the
structure and content of the upcoming Java course updates, rather than to
evoke a particular emotional response. Therefore, it is best categorized as
**neutral**.
```

Fig. 19.2 | Analyzing the sentiment of a transcript.

Markdown-Formatted Responses

The chat completion API often returns responses in **markdown format**, as it did when we ran this example. Markdown is a lightweight formatting language that uses text delimiters to indicate formatting instructions. Various code editors and IDEs can read markdown formatting delimiters and render the text in HTML (webpage) or other formats. The following are examples of markdown formatting that a markdown-enabled tool would format:

- ``` delimits blocks of program code that should be formatted in a fixed-width code font,
- ` delimits code words that should be formatted in a code font inside non-code paragraphs,
- ** delimits text that should be formatted in bold for emphasis
- numbers starting from `1`, followed by a period and a space indicate items in a numbered list, and
- items starting with a `-` followed by a space indicate a bullet in a bullet list.

The website `markdownguide.org` provides a Getting Started guide at

Click here to view code image

```
https://www.markdownguide.org/getting-started/
```

and an overview of markdown syntax at

Click here to view code image

```
https://www.markdownguide.org/basic-syntax/
```

To experiment with markdown, check out the free online markdown editor with side-by-side live content rendering at `https://dillinger.io`.

19.4.3 Accessible Image Descriptions

A key goal of the World Wide Web Consortium (W3C) is to make the web accessible to everyone, regardless of their abilities. To that end, accessible image descriptions are vital for people who are blind or have low vision. As we'll discuss in Section 19.9.7, method `describeImage` in our `OpenAIUtilities` class passes a `"system"` message to an OpenAI model indicating that the model is an expert in creating accessible descriptions of images per the W3C's guidelines.[29]

29. World Wide Web Consortium (W3C), "Web Accessibility Tutorials: Images." Accessed February 25, 2025. `https://www.w3.org/WAI/tutorials/images/`.

Figure 19.3 calls our `describeImage` method to get accessible descriptions of

- a UML activity diagram showing a `for` loop's flow of control (lines 14–19) and
- a photo from a beach in Aruba (lines 24–28).

We split the program into parts for discussion purposes. The program assumes these images are in your user account's `Documents` folder in the subfolder

```
examples/ch19/resources
```

Lines 9–10 create a `Path` to the `resources` folder.

Click here to view code image

```
1  // Fig. 19.3: DescribeImageDemo.java
2  // Getting accessibility descriptions of images.
3  import deitel.openai.OpenAIUtilities;
4  import java.nio.file.Path;
5
6  public class DescribeImageDemo {
7     public static void main(String[] args) throws Exception {
8        // location of resources folder containing sample images
9        Path resourcesPath = Path.of(System.getProperty("user.home"),
10          "Documents", "examples", "ch19", "resources");
11
```

Fig. 19.3 | Getting accessibility descriptions of images.

Getting an Accessible Description of the `for`-Loop UML Diagram

Lines 14–19 call our `OpenAIUtilities` class's `describeImage` method to get an accessible description of Section 4.3's UML activity diagram (`ForLoop.png`) showing the flow of control through a Java `for` loop. This interaction with the chat-completion API uses a multimodal prompt consisting of

- the model (`"gpt-4o"`) the OpenAI API should use to obtain the accessible description of the image,
- a text prompt giving a brief overview of what the image is and
- the `Path` to the image to upload.

Click here to view code image

```
        // get an accessible description of a diagram using gpt-4o
        System.out.println("ACCESSIBLE DESCRIPTION OF A DIAGRAM");
        String diagramDescription =
           OpenAIUtilities.describeImage("gpt-4o", """
              The attached image is a UML activity diagram showing the
              flow of control through a Java 'for' loop with UML
              notes for additional context.""",
              Path.of(resourcesPath.toString(), "ForLoop.png"));
        System.out.printf("%s%n%n", diagramDescription);

```

For Section 4.3's UML activity diagram:
The OpenAI API returned the following chat completion:

Click here to view code image

```
ACCESSIBLE DESCRIPTION OF A DIAGRAM
The image is a highly detailed diagram showing the flow of control within a
logical structure of a Java 'for' loop, using the Unified Modeling Language
(UML) Activity Diagram standards. This diagram visually represents the se-
quence and operations involved in executing a 'for' loop in Java, a common
programming language used for building applications.

### Visual Breakdown:
1. **Diagram Format:** The activity diagram is structured to use standard UML
elements such as nodes, activities, decision points, transitions, and annota-
tions for depicting the logical flow.

2. **For Loop Structure:**
   - **Initialization:** The diagram starts with an initial node, indicating
the start of the logical process. This initial node flows into an activity
node representing the 'for' loop's initialization section (e.g., int i = 0).
   - **Condition Checking:** The next activity depicts the condition that
must be met for the loop to execute. This is shown with a diamond-shaped de-
cision node. It checks if the loop condition is true (e.g., i < 10), indicat-
ing whether the loop should continue or terminate.

   - **Loop Body:** From the true branch of the decision node, the flow pro-
ceeds to an activity node representing the loop body where actions (e.g.,
logic or statements) are executed if the condition is satisfied. These ac-
tions are enclosed within a rectangular activity node.
   - **Iteration:** A control flow arrow then typically points from the loop
body back to a small circular node, symbolizing the iterative aspect of the
loop (e.g., i++), where loop control variables are updated.

   - **Re-checking Condition:** After updating, control returns to the deci-
sion node to re-evaluate the condition.

3. **End of Loop:** If the condition is no longer satisfied, the control flow
```

```
follows the false branch from the decision node, leading to a final node.
This terminal node represents the end of the loop operation.

4. **Annotations:** The diagram may include UML notes or text boxes that pro-
vide additional context or explanations at various stages of the loop for
clarity. These are depicted generally as small rectangles connected to the
relevant part of the diagram by dashed lines.

5. **Flow Arrows:** The flow of control is indicated with arrows that con-
nect nodes and activities, forming a clear path illustrating the iterative
process of the 'for' loop.

6. **Styling and Color:** The diagram would use distinct shapes for decision
nodes (diamond), activity nodes (rounded rectangles), and annotations to
avoid confusion and provide a clear visual means to follow the logic.

Overall, this diagram serves as a tool to understand the execution flow of a
'for' loop in Java, detailing the iterative process in a structured graphi-
cal format. Each component is meticulously laid out to ensure the entire
mechanism is visually and logically represented.
```

Getting an Accessible Description of a Scenic Aruba Beach Photo

Lines 24–28 repeat the process for a scenic Aruba beach photo.

Click here to view code image

```
22          // get an accessible description of a photo using gpt-4o
23          System.out.println("ACCESSIBLE DESCRIPTION OF A PHOTO");
24          String imageDescription =
25             OpenAIUtilities.describeImage("gpt-4o", """
26                   Describe in detail the attached photo of a beach scene,
27                   including every object in the photo.""",
28                Path.of(resourcesPath.toString(), "beach.jpg"));
29          System.out.printf("%s%n%n", imageDescription);
30       }
31    }
```

For the following photo:
The OpenAI API returned the following chat completion:

Click here to view code image

```
ACCESSIBLE DESCRIPTION OF A PHOTO
The photo depicts a serene beach scene under a clear blue sky. Central to the
image is a solitary palm tree with green and yellow fronds, casting a shadow
on the white sandy beach below. Two lounge chairs are placed side by side
```

```
facing the ocean, with one having a blue towel draped over its back.

In the background, the turquoise ocean stretches towards the horizon, where a
few small boats and a sailboat can be seen. On the right side of the image,
there is a wooden pier extending into the water, with a rustic structure fea-
turing a thatched roof. The ocean is calm, with gentle waves lapping against
the shore.

Part of a thatched roof is visible at the top edge of the photo, suggesting a
beach hut or shaded area not fully in view. The overall scene conveys a
peaceful and idyllic beach setting, ideal for relaxation.
```

19.4.4 Language Detection and Translation

Inter-language translation is a challenging problem in natural language processing and AI. With AI advances in large language models, genAIs can translate between languages instantly. Natural language translation has a wide range of uses, including:[30]

30. *ChatGPT* response to "Provide common uses for natural language translation." December 14, 2024. https://chatgpt.com.

- **Cross-language communication:** Facilitating conversations between people speaking different languages.
- **Global business operations:** Translating documents, websites and customer communications for international markets.
- **Tourism:** Helping travelers communicate in foreign countries using.
- **Education:** Enabling students and researchers to access academic content in different languages.
- **Healthcare:** Assisting healthcare providers in understanding patients who speak different languages.
- **Diplomatic relations:** Supporting negotiations and communications between countries, such as at the United Nations.
- **Entertainment:** Translating subtitles, scripts and media for global audiences.

The program in Fig. 19.4 translates text among spoken languages. In the OpenAI APIs, translation is simply a chat request. Our `OpenAIUtilities` class's `translate` method (called in lines 15,

20, 25 and 31; discussed in Section 19.9.8) is similar to the class's `chat` method (Section 19.9.5), but `translate` sends a `"system"` message to the OpenAI model indicating that

- it's an expert in natural language translation, and
- it should `translate` the text specified in our `translate` method's second argument to the language specified in the third argument.

For this demonstration, we translate two English sentences to Spanish and Japanese, then translate the Spanish and Japanese versions back to English. Note that the final results had the same first sentence as the original English text, but the second sentences were different (though similar).

Click here to view code image

```
// Fig. 19.4: TextTranslationDemo.java
// Translating text among spoken languages.
import deitel.openai.OpenAIUtilities;

public class TextTranslationDemo {
   public static void main(String[] args) throws Exception {
      // translate text with OpenAI's gpt-4o model
      System.out.println("TRANSLATION DEMO");
      String english =
         "Today was a beautiful day. Tomorrow looks like bad weather.";
      System.out.printf("ORIGINAL: %s%n%n", english);

      System.out.println("Translating English to Spanish...");
      String spanish =
         OpenAIUtilities.translate("gpt-4o", english, "Spanish");
      System.out.printf("SPANISH: %s%n%n", spanish);

      System.out.println("Translating English to Japanese...");
      String japanese =
         OpenAIUtilities.translate("gpt-4o", english, "Japanese");
      System.out.printf("JAPANESE: %s%n%n", japanese);

      System.out.println("Translating Spanish to English...");
      String spanishToEnglish =
         OpenAIUtilities.translate("gpt-4o", spanish, "English");
      System.out.printf(
         "SPANISH TO ENGLISH: %s%n%n", spanishToEnglish);

      System.out.println("Translating Japanese to English...");
      String japaneseToEnglish =
         OpenAIUtilities.translate("gpt-4o", japanese, "English");
```

```
32          System.out.printf(
33             "JAPANESE TO ENGLISH: %s%n", japaneseToEnglish);
34       }
35    }
```

```
TRANSLATION DEMO
ORIGINAL: Today was a beautiful day. Tomorrow looks like bad weather.

Translating English to Spanish...
SPANISH: Hoy fue un día hermoso. Mañana parece que habrá mal tiempo.

Translating English to Japanese...
JAPANESE: 今日は美しい一日でした。明日は悪天候のようです。

Translating Spanish to English...
SPANISH TO ENGLISH: Today was a beautiful day. Tomorrow it seems there will
be bad weather.

Translating Japanese to English...
JAPANESE TO ENGLISH: Today was a beautiful day. It seems like there will be
bad weather tomorrow.
```

Fig. 19.4 | Translating text among spoken languages.

Generative AI

1 Prompt genAIs, asking why when generative AI translates from one language to another then back to the original language, the final text may not be the same as the original.

19.4.5 Code Generation

Generative AIs can write code in many programming languages, but their code quality can vary substantially from well-written, functioning code that does what your prompt described to code that does not work. So, thoroughly test and verify any code genAIs provide.

Figure 19.5 uses our `OpenAIUtilities` class's `chat` method (lines 20–32) with the `"gpt-4o"` model, a `"system"` message (lines 22–23) telling the model it's an expert Java programmer and a `"user"` message (lines 24–30) discussing the Java program the model should write. Recall from Section 19.4.2 that chat completions often return responses in markdown format, as in this program's output, which includes Java code with an explanation of how it works. The

program the Java code into `RollDie.java` so we can compile and run it later. Lines 38–40 extract the Java code from the response using a regular expression that matches all text between the markdown delimiters `"```java"`, which begins a Java-code block and `"```"`, which terminates a code block. The subexpression `"(.*?)"` captures all the text between the delimiters. `Pattern.DOTALL` enables the pattern to match multiple lines of text. Lines 43–48 write the code into `RollDie.java`—line 46 extracts the text that matched the pattern.[31]

31. Another approach would be to prompt the model to return only the Java code with no additional formatting so you can write it directly to a file.

Click here to view code image

```
1  // Fig. 19.5: CodeGenerationDemo.java
2  // Generating code for a die-rolling simulation.
3  import deitel.openai.OpenAIUtilities;
4  import deitel.openai.OpenAIUtilities.Message;
5  import java.nio.file.Files;
6  import java.nio.file.StandardOpenOption;
7  import java.nio.file.Path;
8  import java.util.List;
9  import java.util.regex.Matcher;
10 import java.util.regex.Pattern;
11
12 public class CodeGenerationDemo {
13    public static void main(String[] args) throws Exception {
14       // path to store Java source-code that's generated
15       Path codePath = Path.of(System.getProperty("user.home"),
16          "Documents", "examples", "ch19", "resources", "outputs");
17
18       // generate Java code with OpenAI's gpt-4o model
19       System.out.println("GENERATE DIE ROLLING SIMULATION CODE");
20       String response = OpenAIUtilities.chat("gpt-4o",
21          List.of(
22             new Message("system",
23                "You are an expert programmer in Java's latest version."),
24             new Message("user", """
25                Create a Java class named RollDie that simulates rolling
26                a die 600,000,000 times as quickly as possible. Summarize
27                the frequencies and display them left-aligned in a
28                two-column format with the column heads "Face" and
29                "Frequency". Avoid loops. In the code, use 3-space indents
30                and restrict each code line to 74 characters or fewer.""")
31          )
32       );
33       System.out.printf("%s%n%n", response);
```

```
        // use regex to extract the code and write it to disk;
        // the Pattern extracts all the text between the delimiters
        // "```java" and "```"
        Pattern pattern =
           Pattern.compile("```java\\s+(.*?)```", Pattern.DOTALL);
        Matcher matcher = pattern.matcher(response);

        // if there was a match, write it into the file RollDie.java
        if (matcher.find()) {
           Files.writeString(
              Path.of(codePath.toString(), "RollDie.java"),
              matcher.group(1).trim(),
              StandardOpenOption.TRUNCATE_EXISTING);
        }
     }
  }
```

````
GENERATE DIE ROLLING SIMULATION CODE
To simulate rolling 600,000,000 dice without using loops, we can leverage
Java's streams and parallel computing capabilities to perform the task
efficiently. Below is a basic implementation of your requirements using Java:

```java
import java.util.concurrent.ThreadLocalRandom;
import java.util.stream.LongStream;
public class RollDie {
   public static void main(String[] args) {
      final int numberOfRolls = 600_000_000;
      final int numberOfSides = 6;

      // Array to hold frequency of each face
      long[] frequencies = new long[numberOfSides];

      // Simulate dice rolls
      LongStream.range(0, numberOfRolls).parallel().forEach(i -> {
         int result = ThreadLocalRandom.current().nextInt(0, numberOfSides);
         synchronized (frequencies) { // Synchronize updates to the array
            frequencies[result]++;
         }
      });

      // Display results
      System.out.printf("%-5s%-10s%n", "Face", "Frequency");

      for (int i = 0; i < numberOfSides; i++) {
         System.out.printf("%-5d%-10d%n", i + 1, frequencies[i]);
      }
   }
}
````
````

```
```

Explanation:

1. **Array Initialization**: We create an array `frequencies` to hold the count of each face result (from 1 to 6).

2. **Dice Roll Simulation**:
   - We use a `LongStream` to simulate 600 million rolls.
   - With `parallel()`, the stream processes elements in parallel, which can significantly speed up processing on multi-core processors.
   - For each roll, a random number between 0 and 5 is generated using `ThreadLocalRandom.current().nextInt(0, numberOfSides)`, representing the dice faces (considering array index starts at 0).

3. **Synchronization**:
   - The updates to the `frequencies` array are synchronized to prevent concurrent modification issues, as multiple threads may attempt to update the same index simultaneously.

4. **Result Display**:
   - Finally, the results are printed with a two-column format ("Face" and "Frequency"), with fields left-aligned.
This approach takes advantage of Java's parallel processing to handle an intensive number of dice rolls efficiently, while avoiding explicit loops by utilizing parallel streams.
```

**Fig. 19.5** | Generating code for a die-rolling simulation.

## Output of Executing `RollDie.java`

After the program wrote the Java code into `RollDie.java`, we compiled and ran it, producing the output below. As in Section 5.4, note that each face appears approximately 100,000,000 times.

```
Face Frequency
1    99983905
2    99988484
3    100016911
4    99994843
5    100019275
6    99996582
```

OpenAI Models for Code Generation

We used the `"gpt-4o"` model for this coding demonstration. At the time of this writing, OpenAI had recently released its advanced
```

reasoning models that excel "at STEM [science, technology, engineering and mathematics], especially math and coding."[32,33] For more advanced programming tasks, consider the reasoning models (or more powerful models as they become available), but keep in mind that these might cost more to use.[34]

32. "Reasoning models." Accessed February 25, 2025. https://platform.openai.com/docs/guides/reasoning/reasoning.

33. "OpenAI o1-mini—Advancing cost-efficient reasoning." Accessed February 25, 2025. https://openai.com/index/openai-o1-mini-advancing-cost-efficient-reasoning/.

34. OpenAI, "Pricing." Accessed February 25, 2025. https://openai.com/api/pricing/.

Other AI Code Capabilities

In addition to writing programs from scratch in various programming languages, many genAIs can also perform coding tasks like:

- **Generating code from diagrams/pseudocode**—Convert UML diagrams or flowcharts to code or convert pseudocode to code.
- **Improving performance**—Improve speed, reduce memory use and parallelize.
- **Debugging**—Locate and fix compilation errors, runtime errors (exceptions) and logic errors.
- **Refactoring**—Simplify code, split code into smaller reusable methods and classes, rename code elements and extract repeated code into a method.
- **Improving readability**—Rename variables, methods and classes and tune existing comments.
- **Code review**—Check code for style, code smells,[35] adherence to clean code guidelines,[36] adherence to SOLID principles[37,38] and adherence to latest idioms.

35. "Code smell." Wikipedia. Wikimedia Foundation. October 13, 2024. https://en.wikipedia.org/wiki/Code_smell.

36. Martin, Robert C., *Clean Code: A Handbook of Agile Software Craftsmanship*. Upper Saddle River, NJ: Addison Wesley, 2008.

37. Martin, Robert C., "Principles and Patterns," Object Mentor, archived September 6, 2015. Accessed February 25, 2025. https://web.archive.org/web/20150906155800/http://

www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf.

38. Watts, Stephen, "SOLID Design Principles Explained: The 5 Class-Design Principles," BMC Blogs, August 26, 2024. Accessed March 8, 2025. https://www.bmc.com/blogs/solid-design-principles/.

- **Commenting and documentation**—Add comments to uncommented code and improve the accuracy of existing comments.
- **Translate code between programming languages**.
- **Analysis**—What does this code do? What's wrong with this code?

19.4.6 Named Entity Recognition (NER) and Structured Outputs

You've seen that generative AIs can determine what a text is about and summarize information. A key aspect of this is a natural language processing (NLP) task called **named entity recognition (NER)**, which attempts to locate and categorize items like dates, times, quantities, places, people, things, organizations and more. Our next example demonstrates a chat request asking the model to perform NER on a body of text. We ask the model to use the OntoNotes Named Entity Tag Set.[39] To make the results more convenient to process, we provide the OpenAI API with the exact JSON response format we wish to receive—known as **structured output**.

39. Linguistic Data Consortium, OntoNotes Release 5.0, LDC2013T19 (Philadelphia: University of Pennsylvania, 2013). Accessed March 8, 2024. https://catalog.ldc.upenn.edu/LDC2013T19.

JSON Output for Named Entities

We'd like the chat-completion API to return its response in the JSON format shown below:

Click here to view code image

```
{
   "entities": [
      {
         "text": "Tim Berners-Lee",
         "tag: "PERSON"
```

```
            },
            {
               "text": "CERN",
               "tag: "ORG"
            },
            ...
         ]
      }
```

This has one `"entities"` attribute in which the value is a JSON array of JSON objects. Each JSON object in the array contains

- a `"text"` attribute representing a named entity identified by the model and
- a `"tag"` attribute indicating that named entity's type.

Calling Class `OpenAIUtilities` class's `chatWithStructuredOutput` Method

Figure 19.6 uses our `OpenAIUtilities` class's `chatWithStructuredOutput` method (Section 19.9.9) to analyze text in the file `web.txt` containing information about the World Wide Web from Chapter 1. This file is located in the `resources` subfolder with this chapter's examples. Method `chatWithStructuredOutput` receives as arguments:

- The OpenAI model to use (`"gpt-4o"`),
- a `"system"` message telling the model it's an expert in named entity recognition and that it should return the results in the specified JSON format,
- a `"user"` message containing the text in which to identify the named entities and
- a `Class` object—`OpenAIUtilities.NamedEntities.class` (line 29) is used by the compiler to create a `Class<OpenAIUtilities.NamedEntities>` object that SimpleOpenAI will use to determine the JSON response format.

The method deserializes the model's JSON response and returns a `NamedEntities` object containing a `List<NamedEntity>` called `entities`. Lines 37–38 get the `List` then use its `forEach` method to output each `NamedEntity`'s `text` and `tag` attributes. The classes `NamedEntities` and `NamedEntity` (discussed in Section 19.9.9) are nested `record` classes in class `OpenAIUtilities`.

Click here to view code image

```
// Fig. 19.6: NamedEntityRecognitionDemo.java
// Identifying named entities and obtaining them as structured outputs.
import deitel.openai.OpenAIUtilities;
import deitel.openai.OpenAIUtilities.Message;
import deitel.openai.OpenAIUtilities.NamedEntities;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.List;

public class NamedEntityRecognitionDemo {
   public static void main(String[] args) throws Exception {
      // load web.txt
      Path transcriptPath = Path.of(System.getProperty("user.home"),
         "Documents", "examples", "ch19", "resources", "web.txt");
      String text = Files.readString(transcriptPath);

      // create a request to obtain named entities
      NamedEntities results = OpenAIUtilities.chatWithStructuredOutput(
         "gpt-4o",
         List.of(
            new Message("system", """
               You are an expert in named entity recognition (NER) using
               the OntoNotes 5.0 NER tag set. Analyze the following text
               and return the named entities in the specified structured
               JSON format. Do not include any additional information in
               your JSON response."""),
            new Message("user", text)
         ),
         OpenAIUtilities.NamedEntities.class
      );

      // display source text
      System.out.printf("SOURCE TEXT:%n%s%n%n", text);

      // display named entities
      System.out.println("ENTITIES:");
      results.entities().forEach(entity ->
         System.out.printf("%s: %s%n", entity.text(), entity.tag()));
   }
}
```

```
SOURCE TEXT:
The World Wide Web (simply called "the web") is a collection of hardware and
software associated with the Internet that allows computer users to locate
and view documents (with various combinations of text, graphics, animations,
audios and videos) on almost any subject. In 1989, Tim Berners-Lee of CERN
(the European Organization for Nuclear Research) began developing HyperText
```

```
Markup Language (HTML)—the technology for sharing information via
"hyperlinked" text documents. He also wrote communication protocols such as
HyperText Transfer Protocol (HTTP) to form the backbone of his new hypertext
information system, which he referred to as the World Wide Web. In 1994,
Berners-Lee founded the World Wide Web Consortium (W3C, https://www.w3.org),
which was devoted to developing web technologies. A key W3C goal is to make
the web universally accessible to everyone regardless of disabilities,
language or culture.

ENTITIES:
World Wide Web: ORG
Internet: ORG
1989: DATE
Tim Berners-Lee: PERSON
CERN: ORG
European Organization for Nuclear Research: ORG
HyperText Markup Language: TECH
HTML: TECH
HyperText Transfer Protocol: TECH
HTTP: TECH
World Wide Web: ORG
1994: DATE
Berners-Lee: PERSON
World Wide Web Consortium: ORG
W3C: ORG
```

Fig. 19.6 | Identifying named entities and obtaining them as structured outputs.

Structured Output Exercises

- Exercise 19.84 asks you to modify this example to perform the NLP task called **parts-of-speech (POS) tagging**, which identifies each word's part of speech in a sentence (e.g., noun, verb, adjective, etc.).
- Exercise 19.36 asks you to modify this example to create a multi-translation app that receives a string and translates it to multiple languages. This is similar to the example in Section 19.4.4 but uses structured outputs for convenient manipulation of the response.

19.5 Speech Synthesis and Speech Recognition

Let's demonstrate the OpenAI API's speech recognition and speech

synthesis capabilities, which transcribe speech to text and synthesize speech from text, respectively. The speech-to-text model can transcribe speech in many spoken languages. The text-to-speech models can produce speech in many languages using nine natural-sounding voices.

19.5.1 English Speech-to-Text for Audio Transcription

Figure 19.7 performs speech recognition on an audio file, transcribing it to text. Our class `OpenAIUtilities`' `speechToText` method (Section 19.9.10) uses the Simple-OpenAI library and **OpenAI's Whisper model** (**`"whisper-1"`**) to transcribe an audio file that's uploaded to the model. Whisper auto-detects the natural language and returns the transcription. Lines 20–21 pass the model's name and the audio file's path to `speechToText`, and line 22 outputs the response. We include a portion of the transcript here, splitting the output into paragraphs for readability. Lines 25–29 write the transcription to the file `WhatsNewInJavaOverview.txt` in the `resources/outputs` subfolder of the chapter's examples folder. The result was nearly 100% accurate, though it spelled our last name, Deitel, in an email address and URL phonetically as `"dytel"`.

Click here to view code image

```
1  // Fig. 19.7: SpeechToTextDemo.java
2  // Transcribing audio files to text.
3  import deitel.openai.OpenAIUtilities;
4  import java.nio.file.Files;
5  import java.nio.file.Path;
6  import java.nio.file.StandardOpenOption;
7
8  public class SpeechToTextDemo {
9     public static void main(String[] args) throws Exception {
10       // get path to resources folder
11       Path resourcesFolder = Path.of(System.getProperty("user.home"),
12          "Documents", "examples", "ch19", "resources");
13
14       // get path to audio file WhatsNewInJavaOverview.m4a
15       Path audioPath = Path.of(resourcesFolder.toString(),
16          "WhatsNewInJavaOverview.m4a");
17
18       // convert speech to text with OpenAI's whisper-1 model
19       System.out.println("Waiting for Transcription...");
```

```
        String transcript = OpenAIUtilities.speechToText(
           "whisper-1", audioPath.toString());
        System.out.printf("TRANSCRIPT:%n%s%n", transcript);

        // write the transcription to a file, overwriting it if it exists
        Path transcriptPath = Path.of(resourcesFolder.toString(),
           "outputs", "WhatsNewInJavaOverview.txt");
        Files.writeString(transcriptPath, transcript,
           StandardOpenOption.CREATE,
           StandardOpenOption.TRUNCATE_EXISTING);
     }
  }
```

```
Waiting for Transcription...
TRANSCRIPT:
Welcome to the What's New in Java lesson. In this lesson I have about two and
a half hours of new video with new examples that will be appearing in our
forthcoming new editions of our Java suite of products and that includes the
new edition of this video course, Java Fundamentals Third Edition, Java How
to Program Twelfth Edition, our Java textbook, and Java for Programmers Fifth
Edition, our corresponding professional book as well.

Throughout these videos I'll be presenting various key Java programming fea-
tures from Java 10 through Java 22 and I'll have a slide on one Java 23 fea-
ture as well that's related to some of the other examples that I'll be
presenting to you here. These videos in the What's New lesson are targeted to
existing lessons in the current product where the features in question could
potentially be used. That does not necessarily reflect where they'll appear
in the new suite of products, but it's a relatively good guess as to where
they'll appear. And as we produce the new versions of these videos, we will
post the new lessons online and eventually this lesson will go away as we
meld the new features into the new presentation as well.
...
```

Fig. 19.7 | Transcribing audio files to text.

Generative AI

1 Prompt genAIs, asking for a list of all the capabilities of OpenAI's Whisper API and to suggest cool Java code examples and demos of its features.

19.5.2 Text-to-Speech

A challenge in natural language processing (NLP) is that there are over 7000 natural languages worldwide.[40] According to the CIA

World Fact Book, the top 11 languages alone account for 44.5% of the world's population.[41] **OpenAI's Audio API** can synthesize speech from text in nine voices (`"alloy"`, `"ash"`, `"coral"`, `"echo"`, `"fable"`, `"onyx"`, `"nova"`, `"sage"` and `"shimmer"`) across 57 languages. OpenAI's text-to-speech documentation provides audio samples for each voice and lists the supported spoken languages:

40. Babbel. "How Many Languages Are There in the World?" Babbel Magazine. Accessed February 18, 2025. https://www.babbel.com/en/magazine/how-many-languages-are-there-in-the-world.

41. Central Intelligence Agency. "World: People and Society." The World Factbook. Accessed February 18, 2025. https://www.cia.gov/the-world-factbook/countries/world/#people-and-society.

Click here to view code image

```
https://platform.openai.com/docs/guides/text-to-speech
```

The voices are optimized for English but also work for other languages. Figure 19.8 uses our `OpenAIUtilities` class's `textToSpeech` method (Section 19.9.11) to synthesize speech using the **OpenAI `"tts-1-hd"` model**. Our method's arguments are as follows:

- The first argument is the model to use—`"tts-1-hd"` returns higher-quality audio than the quicker `"tts-1"` model.
- The second argument is the text to synthesize into speech.
- The third argument (`"onyx"`) is the voice to use.
- The last argument is the filename where the synthesized audio should be stored.

The Audio API returns synthesized speech as an MP3 audio file by default. MP3 is widely supported across platforms and balances file size with audio quality.[42] The Audio API also supports several other audio formats. We write this example's synthesized speech files into the `resources/outputs` folder with this chapter's examples. Simply open the files `english.mp3`, `spanish.mp3` and `japanese.mp3` to listen to the synthesized speech.

42. "MP3." Wikipedia, Wikimedia Foundation. Accessed February 25, 2025. https://en.wikipedia.org/wiki/MP3.

Click here to view code image

```
1  // Fig. 19.8: TextToSpeechDemo.java
2  // Synthesizing speech from text.
```

```
import deitel.openai.OpenAIUtilities;
import java.nio.file.Path;

public class TextToSpeechDemo {
   public static void main(String[] args) throws Exception {
      // get path to the resources folder
      Path outputPath = Path.of(System.getProperty("user.home"),
         "Documents", "examples", "ch19", "resources", "outputs");

      // synthesize speech from text using OpenAI's tts-1-hd model
      System.out.println("TEXT-TO-SPEECH DEMO");
      System.out.println("Synthesizing English speech...");
      String english =
         "Today was a beautiful day. Tomorrow looks like bad weather.";
      OpenAIUtilities.textToSpeech("tts-1-hd", english, "onyx",
         Path.of(outputPath.toString(), "english.mp3").toString());

      System.out.println("Synthesizing Spanish speech...");
      String spanish =
         "Hoy fue un día hermoso. Mañana parece que habrá mal tiempo.";
      OpenAIUtilities.textToSpeech("tts-1-hd", spanish, "onyx",
         Path.of(outputPath.toString(), "spanish.mp3").toString());

      System.out.println("Synthesizing Japanese speech...");
      String japanese = "今日は美しい日でした。明日は悪天候になりそうです。";
      OpenAIUtilities.textToSpeech("tts-1-hd", japanese, "onyx",
         Path.of(outputPath.toString(), "japanese.mp3").toString());
   }
}
```

```
TEXT-TO-SPEECH DEMO
Synthesizing English speech...
Wrote audio to english.mp3
Synthesizing Spanish speech...
Wrote audio to spanish.mp3
Synthesizing Japanese speech...
Wrote audio to japanese.mp3
```

Fig. 19.8 | Synthesizing speech from text.

Speech with Emotion

According to OpenAI's text-to-speech documentation, "There is no direct mechanism to control the emotional output of the audio generated. Certain factors may influence the output audio like capitalization or grammar but our internal tests with these have yielded mixed results."[43] For more details on speech synthesis, visit

43. "Text to speech." Accessed February 25, 2025. https://

platform.openai.com/docs/guides/text-to-speech/overview.

Click here to view code image

```
https://platform.openai.com/docs/guides/text-to-speech/voice-
   options#voice-options
```

Speech Exercises

In Exercise 19.29, you'll convert Fig. 19.8 into a JavaFX app that enables the user to play the synthesized speech. In Exercise 19.30, you'll investigate the `javax.sound.sampled` package's audio-recording features and enhance your solution to Exercise 19.29 with an option to record the user speaking. You'll then transcribe the recorded audio, translate it to a user-specified language and synthesize speech from the translation.

19.6 Image Generation

Figure 19.9 uses the **OpenAI Dall-E API** to generate original images. Lines 18–19, 23–24 and 28–29 call class `OpenAUtilities`'s `image` method (Figure 19.9.12), which requires three arguments:

- the model to use to generate images (`"dall-e-3"`),
- a text prompt describing the image to create and
- the filename where the program should store the image.

The `"dall-e-3"` model currently supports images in the following resolutions:[44]

44. "Image generation." Accessed February 25, 2025. `https://platform.openai.com/docs/guides/images`.

- 1024-by-1024 pixels (the default),
- 1024-by-1792 pixels or
- 1792-by-1024 pixels.

You may request images in standard or high-definition (HD) quality. HD images, which our `image` method requests, provide more detail but take longer to generate. The model returns the image in the **PNG format**, which is widely used on the web for its

lossless compression and ability to represent millions of colors.[45] With **lossless compression**, the image's original information can be restored from its compressed representation.

45. "PNG." Wikipedia. Wikimedia Foundation. Accessed February 25, 2025. https://en.wikipedia.org/wiki/PNG.

Click here to view code image

```
// Fig. 19.9: ImageGenerationDemo.java
// Creating images from text prompts.
import deitel.openai.OpenAIUtilities;
import java.nio.file.Path;

public class ImageGenerationDemo {
   public static void main(String[] args) throws Exception {
      // get path to the resources folder
      Path path = Path.of(System.getProperty("user.home"),
         "Documents", "examples", "ch19", "resources", "outputs");

      // generate images with OpenAI image API and dall-e-3 model
      System.out.println("IMAGE-GENERATION DEMO");
      String prompt = """
         Havanese dog as a Japanese anime character
         in neon colors against a black background""";
      System.out.printf("Generating image for:%n%s%n", prompt);
      OpenAIUtilities.image("dall-e-3", prompt,
         Path.of(path.toString(), "HavaneseAnime.png"));

      prompt = "Havanese dog in the style of Vincent Van Gogh";
      System.out.printf("%nGenerating image for:%n%s%n", prompt);
      OpenAIUtilities.image("dall-e-3", prompt,
         Path.of(path.toString(), "HavaneseVanGogh.png"));

      prompt = "Havanese dog in the style of Leonardo DaVinci";
      System.out.printf("%nGenerating image for:%n%s%n", prompt);
      OpenAIUtilities.image("dall-e-3", prompt,
         Path.of(path.toString(), "HavaneseDaVinci.png"));
   }
}
```

```
IMAGE-GENERATION DEMO
Generating image for:
Havanese dog as a Japanese anime character
in neon colors against a black background
Wrote image to /Users/pauldeitel/Documents/examples/ch19/resources/Havanese-
Anime.png

Generating image for:
Havanese dog in the style of Vincent Van Gogh
```

```
Wrote image to /Users/pauldeitel/Documents/examples/ch19/resources/Havanese-
VanGogh.png

Generating image for:
Havanese dog in the style of Leonardo DaVinci
Wrote image to /Users/pauldeitel/Documents/examples/ch19/resources/Havanese-
DaVinci.png
```

Fig. 19.9 | Creating images from text prompts.

Sample Outputs

The Dall-E API generated the following images in response to this example's prompts. Note the dramatically different looks—not just different colors and styles, but also the dogs' faces and hair arrangements! In our experience, we get significantly different results each time we run this program.

OpenAI, DALL·E Image, generated from the prompt, "Havanese dog as a Japanese anime character in neon colors against a black background" on February 25, 2025, https://platform.openai.com/.

OpenAI, DALL·E Image, generated from the prompt, "Havanese dog in the style of Vincent Van Gogh" on October 25, 2024, https://platform.openai.com/.

OpenAI, DALL·E Image, generated from the prompt, "Havanese dog in the style of Leonardo DaVinci" on October 25, 2024, https://platform.openai.com/.

How Does ChatGPT Ensure Images Are Original?

We prompted ChatGPT with "How do you ensure that the images Dall-E creates are original?" ChatGPT responded as follows:[46]
DALL-E generates images using machine learning models trained on large datasets of publicly available images and text, not by copying or manipulating existing images directly. Here's how originality is maintained:

46. ChatGPT in response to "How do you ensure that the images Dall-E creates are original?" October 14, 2024, https://chatgpt.com/.

- **Generative Process**: DALL-E uses a process called "**diffusion**," where the model starts from random noise and iteratively refines it based on the given text prompt. This ensures that the **output is generated from scratch each time**. The model **doesn't retrieve or edit existing images** but instead **creates entirely new ones** based on patterns it has learned from its training data.
- **No Direct Copying**: DALL-E **doesn't store specific images from its training data**. It learns to understand relationships

between textual descriptions and visual features, but the **outputs are not replicas of any particular image in the dataset**. As a result, the images it creates are novel combinations of visual elements that match the given prompts.

- **Ethical Guidelines**: The developers of DALL-E implement safeguards and policies to ensure the model adheres to ethical standards, such as **not creating content that is offensive, harmful, or violates copyright**. Additionally, it's designed to **avoid creating recognizable likenesses of specific individuals or copyrighted works**, focusing instead on **generating creative, original content**.
- **Diversity in Outputs**: The same prompt can result in different images across multiple generations. This variability demonstrates that **the images aren't pre-generated or stored but are the result of a creative, generative process every time**.
- **Training Data**: The model is trained on large, diverse datasets, which include **publicly available images**, but it **doesn't reproduce them**. Instead, it **learns high-level concepts and relationships between images and text**, which it uses to **create original art in response to new prompts**.

If you're concerned about originality and copyright, it's always advisable to **review the specific use cases for the generated images** and consider any legal implications based on how the images will be used (e.g., **commercial vs. personal use**).

19.7 Video

This section presents an application that uses multiple OpenAI APIs to create closed captions for a video, demonstrates OpenAI's Sora video generator and introduces several other powerful video genAIs.

19.7.1 Generating Video Closed Captions

Our next example is a multimodal application (speech, text and video playback) that:

- uploads a video's audio track,
- transcribes it into a JSON format containing timestamps that can be used with the transcribed text to generate **closed captions** (also called **subtitles**), then
- uses a chat-completion request to generate those closed captions in **VTT (Video Text Track) format**.[47]

47. "WebVTT." Wikipedia. Last modified December 31, 2024. Accessed February 25, 2025. https://en.wikipedia.org/wiki/WebVTT.

The following screen captures contain a couple of this app's generated captions overlaid on the original video.

Figure 19.10 call our `OpenAIUtilities` class's `speechToVTT` method (lines 22–23; discussed in Section 19.9.13) to create the VTT captions corresponding to the audio in the file `ImplicitClass.m4a`. Currently, you cannot upload the video file directly to OpenAI to have it generate closed captions. However, you can perform the following steps, which we've implemented in the `speechToVTT` method:

- Use a tool like QuickTime on macOS or the cross-platform VLC Media Player (`https://www.videolan.org/`) to save the video's audio track as an audio file.
- Pass the audio file to the `"whisper-1"` model to perform a transcription. In this example, `speechToVTT` explicitly asks the model to produce a verbose JSON formatted response, causing the model to return a collection of transcription segments. Each segment contains a piece of the transcription and the timestamps where that piece starts and ends in the audio file.
- Build a `String` containing the segments with their start and end timestamps.
- Submit a chat-completion request with the VTT format requirements, asking the chat API to generate captions in the VTT format for video players. We save this to a VTT file.
- Open the video in an application that supports closed captions and play it. For this example, we used the cross-platform VLC Media Player, which allows you to indicate where the VTT file is located. Like the other outputs in this chapter, we write the VTT file to the `resources/outputs` subfolder of this chapter's examples folder.

Figure 19.10's output contains a portion of the VTT-format closed caption text that was written to the standard output and to the file `ImplicitClass.vtt`.

Click here to view code image

```
// Fig. 19.10: SpeechToVTTDemo.java
// Transcribing the audio track from a video and creating closed captions.
import deitel.openai.OpenAIUtilities;

import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.StandardOpenOption;

public class SpeechToVTTDemo {
   public static void main(String[] args) throws Exception {
      // get path to resources folder
      Path resourcesFolder = Path.of(System.getProperty("user.home"),
         "Documents", "examples", "ch19", "resources");

      // get path to audio file ImplicitClass.m4a
      Path audioPath =
         Path.of(resourcesFolder.toString(), "ImplicitClass.m4a");

      // convert speech to closed captions with OpenAI's whisper-1 model
      System.out.println(
         "Transcribing audio and creating VTT captions file...");
      String captions =
         OpenAIUtilities.speechToVTT("whisper-1", audioPath);
      System.out.printf("CAPTIONS:%n%s%n", captions);

      // output the captions to a .vtt file
      var captionsPath = Path.of(resourcesFolder.toString(),
         "outputs", "ImplicitClass.vtt");
      Files.writeString(captionsPath, captions.trim(),
         StandardOpenOption.CREATE,
         StandardOpenOption.TRUNCATE_EXISTING);
   }
}
```

```
CAPTIONS:

WEBVTT

00:00:00.000 --> 00:00:07.700
In this video, we'll take another look at the JEP 463 capabilities, and in
particular,

00:00:07.700 --> 00:00:12.200
```

```
we're going to demonstrate an implicitly declared class.

00:00:12.200 --> 00:00:16.760
So basically, we're going to have a source code file with no class defini-
tion at all

00:00:16.760 --> 00:00:17.760
in it.

...
```

Fig. 19.10 | Transcribing the audio track from a video and creating closed captions.

19.7.2 Sora

In February 2024, OpenAI announced their **Sora video generator**.[48] Though it was not the first AI video generation tool, Sora sparked a flurry of activity in the genAI video space with many powerful video models released throughout 2024—some free, some with free tiers and some paid. in December 2024, OpenAI released Sora to ChatGPT subscribers[49] for use at

48. "Sora," OpenAI, February 2024. Accessed February 27, 2025. https://openai.com/index/sora/.
49. "Sora Is Here," OpenAI. December 9, 2024. Accessed February 27, 2025. https://openai.com/index/sora-is-here/.

```
https://sora.com
```

At the time of this writing, Sora does not have an API. So, you can use it only via its web interface, which supports creating videos by

- supplying a text-only prompt describing the video to generate,
- supplying an image or video clip on which to base a generated video, optionally providing additional guidance via a text prompt, or
- using the Storyboard tool for more precise control over a video's production.

With the Storyboard, you can place images, video clips and text prompts in individual video frames along a timeline and Sora will use them to guide generation of a video. You also can take a video

you generated previously and “re-cut” it. This opens a generated video in the storyboard so you can edit aspects of it.[50]

50. “Generating Videos on Sora,” OpenAI Help Center. Accessed February 27, 2025. https://help.openai.com/en/articles/9957612-generating-videos-on-sora.

Generating Video from Scratch Via a Text Prompt

Below are some frames of a video Sora generated based on the prompt, “Havanese puppy playfully running around in its back yard.” The full video (`HavanesePuppy.mp4`) is provided in the `resources/sora` folder with this chapter’s examples.

OpenAI, Sora video, generated from the prompt, “Havanese puppy playfully running around in its back yard” on February 27, 2025, https://sora.com/.

Mistakes

As we’ve discussed, genAIs make mistakes and hallucinate, so you need to verify their results. Though video genAIs can create hyper-realistic videos, they sometimes make mistakes. For the next video, we provided Sora with the prompt, “Two havanese puppies, one with black hair and white paws, playfully running around their backyard.” It once again generated realistic video, but notice in the following video frames that

- initially there are two puppies and the blonde one has two tails and no head,
- in the second frame a third puppy’s head appears above the blonde puppy and
- later the blonde puppy completely disappears.

The full video (`DogsWithMistakes.mp4`) is provided in the `resources/sora` folder with this chapter’s examples.

OpenAI, Sora video, generated from the prompt, “Two havanese puppies, one with black hair and white paws, playfully running around their backyard” on February 27, 2025, https://sora.com/.

Generative AI

1 Prompt genAIs for an overview of generating videos with Sora’s storyboard feature.

2 Prompt genAIs for an overview of “re-cutting” a video using Sora’s storyboard feature.

19.7.3 Other Video GenAIs

Many video genAIs are available or under development. In Exercises 19.66–19.69, you'll investigate the following genAIs that each offer unique capabilities. Be sure to check out the cool video demos at each website:

- **Adobe Firefly** (https://firefly.adobe.com) is Adobe's subscription AI service that includes video generation and editing and is integrated into many Adobe image and video creative tools. Firefly now offers a limited free tier.
- **Meta Movie Gen** (https://ai.meta.com/research/movie-gen/) video genAI provides advanced video generation and editing capabilities, currently geared toward storytellers, such as movie directors. At the time of this writing, it's still under development and available only to select Meta partners.
- **Google's Veo 2** (https://deepmind.google/technologies/veo/veo-2/) video genAI claims to be more realistic than other video genAIs due to its level of detail, understanding of physics and range of camera control options. At the time of this writing, it's in early access. You can sign up for the waitlist at

  Click here to view code image

  ```
  https://labs.google/fx/tools/video-fx
  ```

- **InVideo** (https://invideo.io/) automates many aspects of creating professional marketing and educational videos, including scriptwriting, voice overs and more. They provide a free limited tier for experimentation.

Generative AI

1 Prompt genAIs for a list of video genAIs that are free or offer free tiers.

19.8 Moderation

The **OpenAI moderation API**[51] helps developers identify and filter potentially harmful or inappropriate text and image content. The **`"omni-moderation-latest"` model** evaluates text and image inputs for harassment, hate speech, violence, self-harm and sexually explicit content—several of these also have more specific subcategories. For each category and sub-category, the response contains:

51. OpenAI, "Moderation," OpenAI Platform Documentation. Accessed February 26, 2025. `https://platform.openai.com/docs/guides/moderation`.

- a `boolean` indicating whether that category was detected and
- a confidence score from 0 to 1—higher values indicate more offensive content.

For a list of the moderation API's reporting categories, visit:

Click here to view code image

```
https://platform.openai.com/docs/guides/moderation
```

Apps can use the results to ensure user safety and comply with ethical guidelines. For information on OpenAI's usage policies and guidelines, visit:

Click here to view code image

```
https://openai.com/policies/usage-policies/
```

Demonstrating Moderation

Figure 19.11 presents a simple example of using the OpenAI moderation API. The program obtains a prompt from the user, then line 14 calls our `OpenAIUtilities` class's `static` method `checkPrompt` (Section 19.9.14) to evaluate the prompt for offensive content using the moderation API. Method `checkPrompt` returns a Simple-OpenAI `ModerationResult` object. Line 16 calls its `getFlagged` method to check whether the prompt was flagged. If so, lines 21–46 show the individual moderation categories and their scores. Lines 21–22 get a `ModerationResult.Category` object containing the categories' `boolean` values and a `ModerationResult.CategoryScore` object containing the categories scores. Both classes provide identically named *get* methods to obtain the `boolean` values and scores (called in lines 32–46).[52]

52. At the time of this writing (Simple-OpenAI version 3.18.0), both the

`Category` and `CategoryScore` classes contain the misspelled method `getViolencGraphic`, which should be `getViolenceGraphic`. This bug was reported and slated to be fixed in the next Simple-OpenAI release.

Click here to view code image

```
// Fig. 19.11: ModerationDemo.java
// Using the OpenAI moderation API to check for offensive content.
import deitel.openai.OpenAIUtilities;
import java.util.Scanner;

public class ModerationDemo {
   public static void main(String[] args) {
      var input = new Scanner(System.in);

      System.out.print("Enter prompt (or "exit"): ");
      String prompt = input.nextLine();

      while (!prompt.toLowerCase().equals("exit")) {
         var result = OpenAIUtilities.checkPrompt(prompt);

         if (!result.getFlagged()) {
            System.out.println(
               "Prompt not flagged for offensive content");
         }
         else {
            var categories = result.getCategories();
            var scores = result.getCategoryScores();

            // display report
            System.out.println("Offensive content categories & scores:");
            System.out.printf("""
               harassment: %b %.2f; threatening: %b %.2f
               hate: %b %.2f; threatening: %b %.2f
               self-harm: %b %.2f; instructions: %b %.2f; intent: %b %.2f
               sexual: %b %.2f; minors: %b %.2f
               violence: %b %.2f; graphic: %b %.2f%n""",
               categories.getHarassment(), scores.getHarassment(),
               categories.getHarassmentThreatening(),
               scores.getHarassmentThreatening(),
               categories.getHate(), scores.getHate(),
               categories.getHateThreatening(),
               scores.getHateThreatening(),
               categories.getSelfHarm(), scores.getSelfHarm(),
               categories.getSelfHarmInstructions(),
               scores.getSelfHarmInstructions(),
               categories.getSelfHarmIntent(), scores.getSelfHarmIntent(),
               categories.getSexual(), scores.getSexual(),
               categories.getSexualMinors(), scores.getSexualMinors(),
               categories.getViolence(), scores.getViolence(),
```

```
                categories.getViolencGraphic(),
                scores.getViolencGraphic());
        }

        System.out.print("\nEnter prompt (exit to terminate): ");
        prompt = input.nextLine();
      }
    }
}
```

```
Enter prompt (exit to terminate): I would die for an ice cream sundae right now
Prompt not flagged for offensive content

Enter prompt (exit to terminate): I'm going to make them pay.
Offensive content categories & scores:
harassment: false 0.25; threatening: false 0.33
hate: false 0.01; threatening: false 0.00
self-harm: false 0.00; instructions: false 0.00; intent: false 0.00
sexual: false 0.00; minors: false 0.00
violence: true 0.42; graphic: false 0.00

Enter prompt (exit to terminate): exit
```

Fig. 19.11 | Using the OpenAI moderation API to check for offensive content.

Sample Output

To test the moderation API, we began with the statement,

Click here to view code image

```
I would die for an ice cream sundae right now.
```

In this sentence, the word `"die"` is used as hyperbole to emphasize the person's extreme enthusiasm for finding an ice cream sundae immediately.

To cause the moderation API to flag content, we needed a statement that could be interpreted as offensive. We prompted ChatGPT to list some phrases that were not overly offensive but would be likely to be flagged and used the following suggested one:

```
I'm going to make them pay.
```

The moderation API interpreted the statement as offensive, indicating that it tested positive for violence with a score of 0.42.

The output lines containing multiple categories list a main

category and subcategories. For example, the output line

Click here to view code image

```
harassment: false 0.25; threatening: false 0.33
```

shows the `"harassment"` category and its subcategory `"harassment/threatening"`, which we shorthanded as `"threatening"`. Interestingly, the moderation API gave scores of `0.25` and `0.33` for the categories `"harassment"` and `"harassment/threatening"`, but did not flag those as `true`. Each category has a minimum threshold for which it reports `true` but the exact values are not published in the moderation API's documentation.

19.9 Class `OpenAIUtilities`

In this chapter's examples, we focused on each application's genAI concepts, encapsulating the details of using the Simple-OpenAI library to interact with the OpenAI APIs in this section's `OpenAIUtilities` class. We present this class's methods and nested types in the order the examples used them. As discussed in Section 19.3.2, the `libs` subfolder with this chapter's examples contains a JAR file containing the compiled `OpenAIUtilities` class. For those who wish to modify this class, we discuss the IntelliJ IDEA project we used to manage the class and its dependencies in our IntelliJ IDEA Community Edition test-drive at `https://deitel.com/javafp5`. We also provide the project with the chapter's examples.

19.9.1 Builder Design Pattern

Section 18.6.2 introduced the builder design pattern—a common technique for constructing complex objects step-by-step via chained method calls. When working with the Simple-OpenAI Library, you'll use this pattern to

- create a `SimpleOpenAI` object for performing the interactions with the OpenAI APIs and
- to formulate the OpenAI API requests.

Synchronous vs. Asynchronous Execution

In this chapter's examples, the programs issue OpenAI API requests and await their entire responses before outputting the results. The

Simple-OpenAI library supports asynchronous programming. When you submit a request, Simple-OpenAI returns a **`CompletableFuture`**, which is a `Future` (Section 18.7.1) with additional capabilities. Some features of `CompletableFuture` include:

- combining multiple asynchronous tasks then handling their results collectively when they become available,
- providing "callback" methods that are called automatically to process results when a task completes and
- more control over a task's lifecycle, such as manually terminating a task.

Each method in our `OpenAIUtilities` class calls a `CompletableFuture`'s `join` method to synchronously await the results. For more information on `CompletableFuture`s, visit:

Click here to view code image

```
https://docs.oracle.com/en/java/javase/23/docs//api/java.base/java/
   util/concurrent/CompletableFuture.html
https://www.baeldung.com/java-completablefuture
https://dzone.com/articles/20-examples-of-using-javas-
   completablefuture
```

19.9.2 `package` and `import` Statements

Class `OpenAIUtilities` (Fig. 19.12) uses the Simple-OpenAI open-source library to interact with the OpenAI APIs. This class is defined in its own IntelliJ project as a library that we import into this chapter's examples. We split the class into parts for discussion purposes. To enable our apps to use class `OpenAIUtilities`, they must be able to import the class from our custom library. For this reason, we placed this class `OpenAIUtilities` in the custom `deitel.openai` package by using a **`package` statement** (line 4), which must appear in a Java source-code file before any `import` statements. Lines 7–22 import the Simple-OpenAI library classes—we discuss each as we use it.

Click here to view code image

```
1  // Fig. 19.12: OpenAIUtilities.java
2  // Helper methods and record classes to used encapsulate calling
3  // the OpenAI APIs via the Simple-OpenAI Java library.
4  package deitel.openai;
```

```
5
6  import com.fasterxml.jackson.databind.ObjectMapper;
7  import io.github.sashirestela.openai.SimpleOpenAI;
8  import io.github.sashirestela.openai.common.ResponseFormat;
9  import io.github.sashirestela.openai.common.ResponseFormat.JsonSchema;
10 import io.github.sashirestela.openai.common.audio.Voice;
11 import io.github.sashirestela.openai.common.content.ContentPart;
12 import io.github.sashirestela.openai.domain.audio.AudioResponseFormat;
13 import io.github.sashirestela.openai.domain.audio.SpeechRequest;
14 import io.github.sashirestela.openai.domain.audio.Transcription;
15 import io.github.sashirestela.openai.domain.audio.TranscriptionRequest;
16 import io.github.sashirestela.openai.domain.chat.Chat;
17 import io.github.sashirestela.openai.domain.chat.ChatMessage;
18 import io.github.sashirestela.openai.domain.chat.ChatRequest;
19 import io.github.sashirestela.openai.domain.image.ImageRequest;
20 import io.github.sashirestela.openai.domain.moderation.Moderation;
21 import io.github.sashirestela.openai.domain.moderation.ModerationRequest;
22 import io.github.sashirestela.openai.support.Base64Util;
23
24 import java.io.FileOutputStream;
25 import java.net.URL;
26 import java.net.URI;
27 import java.nio.file.Files;
28 import java.nio.file.Path;
29 import java.nio.file.Paths;
30 import java.nio.file.StandardCopyOption;
31 import java.util.List;
32 import java.util.stream.Collectors;
33
34 public class OpenAIUtilities {
```

Fig. 19.12 | Helper methods and a record class to encapsulate calling the OpenAI APIs via the Simple-OpenAI Java library.

19.9.3 Nested `record` Class `Message`

The `record` class `Message` (line 37) groups an OpenAI `role` (`"system"`, `"user"` or `"assistant"`) and its corresponding `content`. We'll turn each `Message` object into an object of a Simple-OpenAI class that represents a message with the specified role. Simple-OpenAI, in turn, will convert these to messages sent to OpenAI's web services with our requests.

Click here to view code image

```
35     // record class used to simplify passing messages containing a role
36     // and corresponding content to this class's chat and translate methods
37     public record Message(String role, String content) {};
```

19.9.4 `SimpleOpenAI` Client Object

An object of the **Simple-OpenAI library's `SimpleOpenAI` class** authenticates with OpenAI using your `OPENAI_API_KEY` and provides methods that invoke OpenAI's web services. Lines 40–41:

- call the **`SimpleOpenAI` `static` method `builder`** to get a **`SimpleOpenAIBuilder` object** for building a `SimpleOpenAI` that is authenticated to use the OpenAI APIs,
- call the **`SimpleOpenAIBuilder`'s `apiKey` method** to specify our `OPENAI_API_KEY`, which we load from the environment variable you created in Section 19.2.4, then
- call the **`SimpleOpenAIBuilder`'s `build` method** to create the `SimpleOpenAI` client object for interacting with the OpenAI APIs.

We store the resulting `SimpleOpenAI` object in a `static` class variable for use by all the `OpenAIUtilities` class's methods.

Click here to view code image

```
39      // create a SimpleOpenAI client object
40      private final static SimpleOpenAI openAI = SimpleOpenAI.builder()
41         .apiKey(System.getenv("OPENAI_API_KEY")).build();
42
```

19.9.5 `chat` Method

`OpenAIUtilities` method `chat` (lines 44–59) receives the OpenAI model to use for a chat completion and a `List<Message>`, builds a chat request, uses it to invoke OpenAI's chat-completion API[53] and returns the chat response as a `String`.

53. OpenAI, *Chat Completions API Reference*, OpenAI Platform. Accessed February 25, 2025. https://platform.openai.com/docs/api-reference/chat.

Click here to view code image

```
43      // perform a chat completion
44      public static String chat(String chatModel, List<Message> messages) {
45         // start building a ChatRequest
```

```
46          var chatRequest = ChatRequest.builder().model(chatModel);
47
48          // add messages to the ChatRequest
49          for (var currentMessage : makeChatMessages(messages)) {
50             chatRequest = chatRequest.message(currentMessage);
51          }
52
53          var request = chatRequest.build(); // complete building the request
54
55          // make the ChatRequest and get the Chat response
56          var chatResponse = openAI.chatCompletions().create(request).join();
57
58          return chatResponse.firstContent(); // return the response text
59       }
60
```

The method operates as follows:

- Line 46 calls the Simple-OpenAI **`ChatRequest` class's `static` `builder` method** to get a **`ChatRequestBuilder` object** for configuring a `ChatRequest`. We call this object's method to add options to the request.
- Next, line 46 calls the **`ChatRequestBuilder` object's `model` method** to set the chat-completion model.
- As discussed in Section 19.4.1, the OpenAI chat endpoint can receive `"system"`, `"user"` and `"assistant"` messages. The Simple-OpenAI library represents these with the **`ChatMessage` class**, which has **`SystemMessage`**, **`DeveloperMessage`**, **`UserMessage`**, **`AssistantMessage` subclasses**. Lines 49–51 continue building the chat request by calling our `makeChatMessages` utility method (Section 19.9.6) to convert our `chat` method's `messages` argument into a `List<ChatMessage>`. For each `ChatMessage`, line 50 calls the **`ChatRequestBuilder`'s `message` method**, which adds a `ChatMessage` to the request and returns the updated `ChatRequest-Builder`. We then assign the update builder to `chatRequest`.
- When all the messages have been added, line 53 calls **`ChatRequestBuilder`'s `build` method** to finish building the `ChatRequest`.
- Line 56 sends the request to OpenAI and awaits its response. First, we call the **`SimpleOpenAI` object's `chatCompletions` method** to get a **`ChatCompletions` object** that encapsulates the details of invoking OpenAI's chat-completion web

service. We submit the `request` by calling the **`ChatCompletions` object's `create` method**, passing `request` as the argument. Method `create` returns a `CompletableFuture<Chat>` from which we'll obtain a Simple-OpenAI `Chat` object containing the response, once it's available. Calling the `CompletableFuture<Chat>`'s `join` method causes the caller to wait for the completed response before proceeding.

- Finally, line 58 calls the **`Chat` object `chatResponse`'s `firstContent` method** to get and return the `String` representing the completed chat response.

Mapping Simple-OpenAI Classes and Methods to OpenAI API Documentation

The Simple-OpenAI classes and method calls used in our `chat` method map to the options discussed in the Request body section of the OpenAI chat completion API documentation at

Click here to view code image

```
https://platform.openai.com/docs/api-reference/chat/create
```

For example,

- the `model` method call (line 46) specifies the value of the chat API's `model` parameter and
- each `message` call (line 50) adds a message to the chat API's `messages` parameter.

A benefit of writing Simple-OpenAI code in an IDE like IntelliJ is that, as you enter code, the IDE can tell you the methods you can call and what their parameters are. We demonstrate this in our IntelliJ IDEA Community Edition test-drive at

```
https://deitel.com/javafp5
```

19.9.6 Helper Method `makeChatMessages`

Our helper method `makeChatMessages` (lines 63–77) converts a `List<Message>` into a `List<ChatMessage>` containing `SystemMessage`, `UserMessage` and `AssistantMessage` objects, based on each `Message`'s role (`"system"`, `"user"` or `"assistant"`). The

stream pipeline in lines 66–76 maps each `Message` object to the appropriate `ChatMessage` subclass object (lines 67–75). Line 76 converts these into a `List`, which `makeChatMessages` returns.

Click here to view code image

```
61      // utility method to convert our Message record objects into
62      // SystemMessage, UserMessage and AssistantMessage objects
63      private static List<ChatMessage> makeChatMessages(
64         List<Message> messages) {
65         // convert each Message record to a ChatMessage subclass object
66         return messages.stream()
67            .map(m -> switch(m.role().toLowerCase()) {
68               case "system" ->
69                  ChatMessage.SystemMessage.of(m.content());
70               case "user" -> ChatMessage.UserMessage.of(m.content());
71               case "assistant" ->
72                  ChatMessage.AssistantMessage.of(m.content());
73               default -> throw new IllegalArgumentException(
74                  "invalid role: " + m.role());
75            })
76            .toList();
77      }
78
```

Nested Classes

Classes `SystemMessage`, `UserMessage` and `AssistantMessage` are `public static` nested classes in class `ChatMessage`. A **`static` nested class** is defined inside another class and preceded by the `static` keyword. Such classes are often used when the nested class is closely related to its enclosing class. In this case, the nested classes are also `ChatMessage` subclasses. So, objects of these classes are valid wherever `ChatMessage`s are required. Each of these nested subclasses has an `of` method that receives a `String` representing the message's content and returns a new object of the corresponding `ChatMessage` subclass.

19.9.7 `describeImage` Method

`OpenAIUtilities` method `describeImage` (lines 80–104) uses the OpenAI chat API's vision capabilities to create accessible image descriptions for people who are blind or have low vision. This method's `"system"` message (lines 85–89) directs the chat API to adhere to the accessible image-description guidelines provided by the World Wide Web Consortium (W3C).[54] Method `describeImage`

requires three arguments:

54. W3C Web Accessibility Initiative, “Images,” Web Accessibility Tutorials. Last modified February 8, 2022. Accessed February 25, 2025. https://www.w3.org/WAI/tutorials/images/.

- the chat model that should create the accessible image description,
- a prompt that will be supplied as a `"user"` message to the chat model—used in Section 19.4.3 to provide the model with a brief description of an image—and
- a `Path` to the image that will be uploaded.

OpenAI’s models can also receive the URL of an image to analyze, though we do not support that in our `describeImage` method.

Click here to view code image

```
79      // perform a chat completion
80      public static String describeImage(
81         String chatModel, String prompt, Path imagePath) {
82         // start building a ChatRequest for accessible image descriptions
83         var chatRequest = ChatRequest.builder()
84            .model(chatModel)
85            .message(ChatMessage.SystemMessage.of("""
86               You are an expert at creating accessible descriptions
87               of images per the World Wide Web Consortium's (W3C's)
88               guidelines. Given an image, explain it in detail for
89               people who are blind or have low vision."""))
90            .message(ChatMessage.UserMessage.of(
91               List.of(
92                  ContentPart.ContentPartText.of(prompt),
93                  ContentPart.ContentPartImageUrl.of(
94                     ContentPart.ContentPartImageUrl.ImageUrl.of(
95                        Base64Util.encode(imagePath.toString(),
96                           Base64Util.MediaType.IMAGE))))))
97            .build();
98
99         // make the ChatRequest and get the Chat response
100        var chatResponse =
101           openAI.chatCompletions().create(chatRequest).join();
102
103        return chatResponse.firstContent(); // return the response text
104     }
105
```

Method `describeImage` is similar to `chat` (Section 19.9.5). The key difference is in lines 90–96 of creating the `ChatRequest`. To

analyze an image, the OpenAI chat API requires a JSON array of messages containing a text prompt and the URL of an image to analyze. If the image is a local file on your system, as in Section 19.4.3's example, the URL must have the format:

Click here to view code image

```
" base64EncodedImageString"
```

where *imageFormat* in our example is `png` or `jpeg`, depending on which image we send, and *base64EncodedImageString* is a `String` containing the image's Base64-encoded data (introduced in Section 7.9). Lines 90–96 create a JSON array of messages:

- When you create a `ChatMessage` with a `List` as an argument, Simple-OpenAI converts the `List` to a JSON array—lines 91–96 create a `List` containing two Simple-OpenAI **`ContentPartS`**, each representing a message in the JSON array.
- The **`ContentPart` nested class `ContentPartText`** (line 92) creates a message for the required text prompt.
- The **`ContentPart` nested class `ContentPartImageUrl`** (lines 93–96) creates a message containing the required Base64-encoded image. The Simple-OpenAI utility class **`Base64Util`'s `encode` method** (lines 95–96) receives as arguments the image's location on your system and the kind of data being encoded—`Base64Util.MediaType.IMAGE` indicates the data is an image. The kind of data is embedded in the Base64 encoding so the OpenAI API can decode it correctly.

19.9.8 `translate` Method

`OpenAIUtilities` method `translate` (lines 107–122) uses the chat model specified in its first argument to translate the `String` in its second argument to the language in its third argument, then returns the translated `String`. Text translation is also accomplished via standard chat completions. Lines 111–116 build the `ChatRequest`:

- The `SystemMessage` (`"system"` role) in lines 112–114 tells the chat model how to operate. In this case, the chat is a translation expert that will translate text to the `targetlanguage`.
- The `UserMessage` (`"user"` role) in line 115 provides the text

to translate.

Line 119 submits the request to OpenAI and awaits its response. Then, line 121 returns the result as a `String`.

Click here to view code image

```
106     // perform a chat completion for translation
107     public static String translate(
108        String chatModel, String textToTranslate, String targetLanguage) {
109
110        // build a ChatRequest to translate a String
111        var chatRequest = ChatRequest.builder().model(chatModel)
112           .message(ChatMessage.SystemMessage.of(String.format("""
113              You are an expert in natural language translation.
114              Translate the following text into %s.""", targetLanguage)))
115           .message(ChatMessage.UserMessage.of(textToTranslate))
116           .build();
117
118        // create and submit the ChatRequest and wait for the Chat response
119        Chat response = openAI.chatCompletions().create(chatRequest).join();
120
121        return response.firstContent(); // return the translation
122     }
123
```

19.9.9 `chatWithStructuredOutput` Method

As discussed in Section 19.4.6, OpenAI chat completions can return structured outputs containing data in the format you specify. This section discusses the `record` classes and method `chatWithStructuredOutput` we used in Section 19.4.6's named entity recognition (NER) example.

`record` Classes `NamedEntity` and `NamedEntities`

Lines 126–127 define the `record` classes we used in our named entity recognition example (Section 19.4.6) to represent the structured output format:

- Our `NamedEntity record` class represents one named entity as two `String`s—the entity's `text` and corresponding `tag`.
- Our `NamedEntities record` class represents a `List` of `NamedEntity` objects.

Click here to view code image

```
124     // record classes to define the structure of a JSON response
125     // containing results of named entity recognition
126     public record NamedEntity(String text, String tag) {}
127     public record NamedEntities(List<NamedEntity> entities) {}
128
```

Method `chatWithStructuredOutput`

`OpenAIUtilities` method `chatWithStructuredOutput` (lines 131–160) is not specific to named entity recognition. It's a generic method that receives as arguments:

- the chat model that will process the chat request,
- a `List<Message>` specifying the messages to pass in the chat request, and
- a `Class<T>` representing the class that will be used to determine the structured output format. In Section 19.4.6, this argument was specified as

  Click here to view code image

  ```
  OpenAIUtilities.NamedEntities.class
  ```

  which the Java compiler converts to an object of type

  Click here to view code image

  ```
  Class<OpenAIUtilities.NamedEntities>
  ```

The `chatWithStructuredOutput` method returns an object of the class represented by the type parameter `T`—for Section 19.4.6's example an `OpenAIUtilities.NamedEntities` object. When calling `chatWithStructuredOutput`, include a `"system"` message telling the chat request to return a JSON response in the "specified format" (determined by lines 134–138) and nothing else; otherwise, the chat response may nest your desired JSON response in a larger JSON object containing additional information.

Click here to view code image

```
129     // generic method that performs a chat completion and
130     // returns structured output
131     public static <T> T chatWithStructuredOutput(String chatModel,
132        List<Message> messages, Class<T> jsonStructure) throws Exception {
133        // set up JSON schema for structured output based on jsonStructure
134        var schema = ResponseFormat.jsonSchema(JsonSchema.builder()
```

```
135             .name(jsonStructure.getSimpleName()) // schema name
136             .schemaClass(jsonStructure) // schema Java class
137             .build()
138         );
139
140         // start building a ChatRequest that returns structured output
141         var chatRequest = ChatRequest.builder()
142             .model(chatModel)
143             .responseFormat(schema);
144
145         // add messages to the ChatRequest
146         for (var currentMessage : makeChatMessages(messages)) {
147             chatRequest = chatRequest.message(currentMessage);
148         }
149
150         var request = chatRequest.build(); // complete building the request
151
152         // make the ChatRequest and get the Chat response
153         Chat response = openAI.chatCompletions().create(request).join();
154
155         // ObjectMapper for deserializing JSON into a NamedEntities object
156         var mapper = new ObjectMapper();
157
158         // return the structured response
159         return mapper.readValue(response.firstContent(), jsonStructure);
160     }
161
```

Determining the Response Format

Section 19.4.6 presented sample JSON containing the expected response format. Lines 134–138 build a **Simple-OpenAI `ResponseFormat` object** representing the structured output format. **`ResponseFormat` `static` method `jsonSchema`** creates a `ResponseFormat` based on its argument—a **Simple-OpenAI `JsonSchema` object**. Lines 134–138, use a **Simple-OpenAI `JsonSchemaBuilder` object** to determine the JSON format:

- Line 135 calls **`JsonSchemaBuilder` method `name`** to specify the class name to which the schema corresponds. **`Class` method `getSimpleName`** returns the unqualified class name of the class represented by the `jsonStructure` argument.
- Line 136 calls **`JsonSchemaBuilder` `schemaClass` method** to infer the JSON structure we'd like to receive in the OpenAI chat response. Method `schemaClass` inspects its argument's class, `NamedEntities`, to determine that it contains a `List<NamedEntity>` called `entities`. This will result in a key-

value pair in the JSON response with the key `"entities"` and a JSON array as the value. It also infers from the `List<NamedEntity>` that each element in the JSON array contains the strings `text` and `tag`. So, each named entity will be represented as a JSON object with key-value pairs having the keys `"text"` and `"tag"`.

- Finally, line 137 builds the `JsonSchema`.

Setting Up the Chat Request

Lines 141–150 set up the chat request similarly to lines 46–53 of method `chat`. The key difference is adding the `JsonSchema` to the request by calling **`ChatRequestBuilder` method `responseFormat`**.

Executing the Request and Processing the Response

Line 153 issues the request using the same technique shown previously. Lines 156 and 159 use a Jackson library `ObjectMapper` to deserialize the JSON response (as discussed in Section 11.5.2) and return an object of the class represented by the generic type `T`. For the named entity recognition example in Section 19.4.6, `chatWithStructuredOutput`'s return value will be a `NamedEntities` object containing a `List<NamedEntity>`.

19.9.10 `speechToText` Method

`OpenAIUtilities` method `speechToText` (lines 163–174) gets a transcription of an audio file. It receives an OpenAI audio-transcription model and a `String` representing the file's location, builds a **Simple-OpenAI `TranscriptionRequest`** (lines 165–168) to get the audio file's transcription, uses the request to invoke OpenAI's Audio API[55] and returns a `String` containing the transcription:

55. OpenAI, *Audio API Reference*, OpenAI Platform. Accessed February 25, 2025, https://platform.openai.com/docs/api-reference/audio.

- Line 165 calls the **`TranscriptionRequest` `static` method `builder`** to get a **`TranscriptionRequestBuilder` object** for configuring the request.
- Line 166 calls the **`TranscriptionRequestBuilder` object's `file`** method to specify the audio file's `Path`.
- Line 167 calls the **`TranscriptionRequestBuilder` object's**

model method to specify the model that will transcribe the audio.

- Line 168 calls the **TranscriptionRequestBuilder object's build** method to create the `TranscriptionRequest`.

Click here to view code image

```
162    // convert audio to text transcription
163    public static String speechToText(String model, String audioPath) {
164       // build a TranscriptionRequest
165       var audioRequest = TranscriptionRequest.builder()
166          .file(Paths.get(audioPath))
167          .model(model)
168          .build();
169
170       // submit TranscriptionRequest and wait for Transcription
171       Transcription transcription =
172          openAI.audios().transcribe(audioRequest).join();
173       return transcription.getText(); // return transcription text
174    }
175
```

Lines 171–172 initiate the transcription request:

- First, we call the **SimpleOpenAI object's audios method** to get a Simple-OpenAI **Audios object** encapsulating the details of invoking OpenAI's audio-transcription web service.
- The **Audios object's transcribe method** receives a `TranscriptionRequest` and returns a `CompletableFuture<Transcription>`, enabling the program to obtain the OpenAI response once it becomes available.
- Again, the `join` method call tells the program not to continue until the complete response is received.

Line 173 returns the transcription.

19.9.11 `textToSpeech` Method

`OpenAIUtilities` method `textToSpeech` (lines 177–192) receives `String`s representing an OpenAI speech-synthesis model, the text to synthesize into speech, the voice used to speak the text and the file in which to store the audio. It sends a speech-synthesis request to the OpenAI Audio API and writes the audio response into the specified file.

Click here to view code image

```
176     // synthesize speech from text
177     public static void textToSpeech(String model, String text,
178        String voice, String filename) throws Exception {
179        // build a SpeechRequest
180        var speechRequest = SpeechRequest.builder()
181           .model(model)
182           .input(text)
183           .voice(Voice.valueOf(voice.toUpperCase()))
184           .build();
185
186        // submit SpeechRequest and wait for audio, then save audio to file
187        try (var audioStream = openAI.audios().speak(speechRequest).join();
188             var audioFile = new FileOutputStream(filename)) {
189           audioFile.write(audioStream.readAllBytes());
190           System.out.printf("Wrote audio to %s%n", filename);
191        }
192     }
193
```

Lines 180–184 build a **Simple-OpenAI `SpeechRequest`**:

- Line 180 calls the Simple-OpenAI **`SpeechRequest` class's `static builder` method** to get a **`SpeechRequestBuilder` object** for configuring the `SpeechRequest`.
- Line 181 calls the **`SpeechRequestBuilder` object's `model` method** to set the text-to-speech model.
- Line 182 calls the **`SpeechRequestBuilder` object's `input` method** to specify the input to the model—that is, the text the model will synthesize into speech.
- Line 183 calls the **`SpeechRequestBuilder` object's `voice` method** to set the voice the text-to-speech model will use. The `Voice` enumeration provides constants for each currently supported voice. We use the `enum`'s `valueOf` method to get the correct constant based on our method's voice argument.
- Line 184 builds the `SpeechRequest`.

Lines 187–191 submit the `SpeechRequest` and write its response to the specified file:

- As with audio transcription, we call the `SimpleOpenAI` object's `audios` method to get a Simple-OpenAI `Audios` object, which also provides access to the OpenAI text-to-

speech capabilities. The **`Audios` object's `speak` method** (line 187) receives a `SpeechRequest` and returns a `CompletableFuture<InputStream>`, enabling the program to obtain the OpenAI response once it becomes available. We call `join` to await the complete response.

- Next, line 188 creates a byte-based `FileOutputStream` that we'll use to write the audio bytes into the specified file.
- Line 189 calls the **`InputStream`'s `readAllBytes` method** to get the complete audio from the response, then writes the bytes into the file using **`FileOutputStream` method `write`**.

When the `try`-with-resources statement completes, both streams are closed automatically.

19.9.12 `image` Method

`OpenAIUtilities static` method `image` (lines 195–217) uses OpenAI's image-generation API[56] to generate an image based on a text prompt. The method receives the name of an image-generation model, a prompt describing the image to generate and a `Path` in which to store the generated image. The method then creates an image-generation request, uses it to invoke the image-generation API and stores the resulting image in the file specified by the `Path` argument.

56. OpenAI, *Images API Reference*, OpenAI Platform. Accessed February 25, 2025, https://platform.openai.com/docs/api-reference/images.

Click here to view code image

```
194    // generate an image with Dall-E
195    public static void image(
196       String model, String prompt, Path imagePath) throws Exception {
197       // build an ImageRequest with the specified model and prompt
198       var imageRequest = ImageRequest.builder()
199          .model(model)
200          .prompt(prompt)
201          .quality(ImageRequest.Quality.HD)
202          .build();
203
204       // submit ImageRequest and wait for image
205       var imageResponse = openAI.images().create(imageRequest).join();
206
207       // get image's URL
208       String urlString = imageResponse.getFirst().getUrl();
209       URL url = URI.create(urlString).toURL();
```

```
210
211         // save image to PNG file
212         try (var inputStream = url.openStream()) {
213            Files.copy(inputStream, imagePath,
214               StandardCopyOption.REPLACE_EXISTING);
215            System.out.printf("Wrote image to %s%n", imagePath);
216         }
217      }
218
```

Lines 198–202 build a **Simple-OpenAI `ImageRequest`**:

- Line 198 calls the Simple-OpenAI **`ImageRequest` class's `static builder` method** to get an **`ImageRequestBuilder` object** to configure the `ImageRequest`.
- Line 199 calls the **`ImageRequestBuilder` object's `model` method** to set the image-generation model.
- Line 200 calls the **`ImageRequestBuilder` object's `prompt` method** to specify the text prompt describing the image to generate.
- Line 201 calls the **`ImageRequestBuilder` object's `quality` method** to set the image quality to high definition (`ImageRequest.Quality.HD`)—the default is standard definition (`STANDARD`).
- Line 202 builds the `ImageRequest`.

This image request produces one generated image (the default). You can request up to 10 images by calling the **`ImageRequestBuilder` object's `n` method** (short for "number") with an integer argument.

Lines 205–216 submit the `ImageRequest` and write the generated image to the specified file:

- Line 205 calls the **`SimpleOpenAI` object's `images` method** to get a **Simple-OpenAI `Images` object**, which provides access to the OpenAI image-generation and editing capabilities. The **`Images` object's `create` method** receives an `ImageRequest` and returns a `CompletableFuture<List<Image>>`, enabling the program to obtain the list of generated images. We call `join` to await the complete response.
- Line 208 calls the `List` method `getFirst` to get the first (and in this case, only) `Image`, then calls the `Image`'s `getUrl` method to get a `String` URL representing the generated image on OpenAI's servers.

- Line 209 converts the `String` URL into a `URL` object.
- Line 212 calls **`URL` method `openStream`** to get a byte-based `InputStream` from which we can read the image's bytes.
- Line 213–214 uses the `Files` class's `static` method `copy` to read the bytes from the first argument's `InputStream` and write them to the second argument's `Path`.

The `InputStream` is closed when the `try`-with-resources statement completes.

19.9.13 `speechToVTT` Method

`OpenAIUtilities` `static` method `speechToVTT` (lines 220–263) receives an OpenAI audio-transcription model name and a `Path` to an audio file, then

- uses a `TranscriptionRequest` to transcribe the audio into segments with timestamps,
- uses a `ChatRequest` to format the transcription into WebVTT format (discussed in Section 19.7.1) for use as closed captions, and
- returns the WebVTT-formatted transcription.

Click here to view code image

```
219    // convert audio track of a video into closed captions
220    public static String speechToVTT(String model, Path audioPath) {
221       // build a TranscriptionRequest that returns JSON containing the
222       // transcription segments and their start/end times
223       var audioRequest = TranscriptionRequest.builder()
224          .file(audioPath)
225          .model(model)
226          .responseFormat(AudioResponseFormat.VERBOSE_JSON)
227          .build();
228
229       // submit TranscriptionRequest and wait for Transcription
230       Transcription transcription =
231          openAI.audios().transcribe(audioRequest).join();
232
233       // get the transcript's segments
234       List<Transcription.Segment> segments = transcription.getSegments();
235
236       // create a String representation of the transcription's
237       // segments with their start and end times
```

```
238         String transcriptionText = segments.stream()
239            .map(segment -> String.format("Start: %f, End: %f, Text: %s",
240               segment.getStart(), segment.getEnd(), segment.getText()))
241            .collect(Collectors.joining("\n"));
242
243         // issue a chat request to convert the audio
244         // segment transcriptions into VTT format
245         var request = ChatRequest.builder()
246            .model("gpt-4o")
247            .message(ChatMessage.SystemMessage.of("""
248               You are a subtitle formatting assistant. Take the following
249               transcription data and format it into VTT (WebVTT) format.
250               Timestamps must have the hh:mm:ss.mmm format required by
251               WebVTT. Include the required WEBVTT header at beginning of
252               the file. Output the VTT content only."""))
253            .message(ChatMessage.UserMessage.of(transcriptionText))
254            .build();
255
256         // make the ChatRequest and get the response
257         var response = openAI.chatCompletions().create(request).join();
258
259         // remove markdown code-block delimiters (```) at beginning
260         // and end of response and return VTT-formatted captions
261         var captions = response.firstContent();
262         return captions.substring(3, captions.length() - 3);
263      }
264
```

The `speechToVTT` method operates as follows:

- Lines 223–227 build an audio-transcription request, as shown previously. The key difference is in line 226 where we call **`TranscriptionRequestBuilder` method `responseFormat`** with **`AudioResponseFormat.VERBOSE_JSON`** as the argument. This tells the transcription model to return audio segments with time-stamps in JSON format. Simple-OpenAI converts the audio segments to objects of class `Transcription.Segment` that are stored in the `Transcription` response.
- Lines 230–231 submit the `TranscriptionRequest`.
- Line 234 gets a `List<Transcription.Segment>` by calling the **`Transcription`'s `getSegments` method**.
- Lines 238–241 use a stream pipeline to map each `Segment` to a formatted `String` containing the `Segment`'s start time, end time and text, then collects these formatted `String`s into a single `String` with each segment separated by a newline from the next.

- Lines 245–257 build and execute a chat-completion request with a `"system"` message (lines 247–252) telling the model how to format the data supplied in the `"user"` message (line 253).
- Finally, lines 261–262 get the content of the chat response. OpenAI returns the formatted response in markdown format, so line 262 removes the ``` ``` ``` delimiters before returning the WebVTT formatted transcription.

19.9.14 `checkPrompt` Method

Method `checkPrompt` (lines 266–276) uses the OpenAI moderation API to check its argument's text for offensive content.

Click here to view code image

```
265     // send text to OpenAI moderation API to check for offensive content;
266     public static Moderation.ModerationResult checkPrompt(String prompt) {
267        // build a ModerationRequest
268        var moderationRequest = ModerationRequest.builder()
269           .model("omni-moderation-latest") // OpenAI recommended model
270           .input(List.of(prompt))
271           .build();
272
273        // submit the request and return the result
274        var result = openAI.moderations().create(moderationRequest).join();
275        return result.getResults().getFirst();
276     }
277  }
```

Lines 268–274 build then execute a **Simple-OpenAI `ModerationRequest`**:

- Line 268 calls the Simple-OpenAI **`ModerationRequest` class's `static builder` method** to get a **`ModerationRequestBuilder` object** to configure the request.
- Line 269 calls the **`ModerationRequestBuilder` object's `model` method** to set the image-generation model.
- Line 270 calls the **`ModerationRequestBuilder` object's `input` method** to specify a list of `String`s to send to the moderation API. In this method, we send one prompt as input, but the `List` may contain multiple items, some of which can be URLs of online images or Base64-encoded images being uploaded from the user's computer.

- Line 271 builds the `ImageRequest`.
- Line 274 calls the **`SimpleOpenAI` object's `moderations` method** to get a **Simple-OpenAI `Moderations` object**, which provides access to the OpenAI moderation API. The **`Moderations` object's `create` method** receives a `ModerationRequest` and returns a `CompletableFuture<Moderation>`, enabling the program to obtain the moderation results when they are available. We call `join` to await the complete response.
- Line 275 gets the first `ModerationResult` from the `List<ModerationResult>` returned by `Moderation` method `getResults`. Each `ModerationResult` contains a `boolean` indicating whether the items was flagged for offensive content, a collection of categories with `boolean` values indicating which were flagged and a corresponding collection of scores from 0 (not offensive) to 1 (completely offensive). We extracted and displayed those results in Fig. 19.11.

19.10 Wrap-Up

In this chapter, we built API-based Java genAI applications that interacted with OpenAI's genAI models via the Simple-OpenAI Java library. We introduced the concept of multi-modal genAIs that process and generate text, images, audio and video content. You learned about several OpenAI genAI models for text generation, speech recognition, speech synthesis, image creation and moderation.

We discussed setting up a Java environment to work with OpenAI APIs using the Simple-OpenAI library. You learned how to obtain an OpenAI developer account, generate an API key and store it securely on your system. We mentioned an IntelliJ-based Maven project setup that simplifies dependency management and compilation of Java applications using the Simple-OpenAI Java library. More details on that project are provided in our IntelliJ IDEA Community Edition test-drive at

```
https://deitel.com/javafp5
```

We presented many genAI apps, starting with several text-based applications. You learned how to generate text using OpenAI's chat-completion models and explored genAI text capabilities such as summarization, sentiment analysis, accessible image descriptions,

language detection and translation, code generation and named entity recognition (NER). In the NER example, we demonstrated the OpenAI chat model's ability to return structured JSON output.

We used OpenAI's Whisper model to perform speech recognition, transcribing audio to text, and their TTS model to synthesize natural-sounding speech in multiple languages. We generated original images from text prompts using OpenAI's Dall-E API. We discussed OpenAI's Sora video generator, which did not have an API at the time of this writing, and introduced several other genAI video services. We used OpenAI's moderation tools to analyze and filter harmful content in text and discussed that moderation can also be used for images.

In our `OpenAIUtilities` class discussion, we explored invoking OpenAI's APIs using the free, open-source Simple-OpenAI library, which encapsulates the details of working with OpenAI's web services. We discussed each of our `OpenAIUtilities` class's `static` helper methods and nested classes that we used in our genAI applications. The next chapter introduces relational databases and Structured Query Language (SQL) then demonstrates using JDBC and SQL to interact with SQLite relational databases.

GenAI Prompt Exercises

19.1 ***(Free GenAI APIs)*** Prompt genAIs to list free genAI APIs or ones that have a free tier or free trial and compare their features to paid versions from those sources.

19.2 ***(Suggest Multimodal Apps)*** Prompt genAIs to tell you the kinds of apps you can build with multimodal genAI inputs and outputs.

19.3 ***(Drug Discovery and Development)*** Prompt genAIs to research how genAI is being used to expedite drug discovery and development.

19.4 ***(Personalized Medicine)*** Prompt genAIs to research how genAI is being used to personalize medicine.

19.5 ***(Medical Imaging Analysis)*** Prompt genAIs to research how genAI is being used to improve medical imaging analysis.

19.6 ***(What Can GenAIs Do?)*** Prompt genAIs to list the kinds of tasks they can perform.

19.7 ***(What Can GenAIs Do with Java Code?)*** Prompt genAIs to list

the tasks they can perform with Java code.

19.8 ***(Research: AI-Generated-Text Detection Tools)*** Prompt genAIs to research detection tools for genAI-generated text, how these tools work, their accuracy and the ethical issues surrounding using such imperfect tools to detect students submitting AI-generated work.

19.9 ***(Research: Google Gemini Free Tier)*** Prompt genAIs to research the Google Gemini API's free tier at `https://ai.google.dev/pricing` and compare the API's capabilities with those you learned in this chapter.

19.10 ***(Research: Hugging Face Free Tier)*** Prompt genAIs to research the Hugging Face API's free tier at `https://huggingface.co/pricing` and compare the API's capabilities with those you learned in this chapter.

19.11 ***(Research: Meta Llama)*** Prompt genAIs to research Meta's free Llama genAI models that you can download from `https://www.llama.com/` and run locally. Compare their models' capabilities with those you learned in this chapter.

19.12 ***(Research: GenAI in Education)*** Prompt genAIs to research ways in which genAI can be used in education by students and instructors.

19.13 ***(GenAI Recipes)*** Prompt genAIs with what's in your refrigerator and cupboard, asking them to give you recipes for what you can cook with these on-hand items.

Projects

Text-Based Apps

19.14 ***(Indexing a Document)*** Create a genAI app that enables a user to upload a Microsoft Word or Google document to the OpenAI chat API, asking for an index of the document's key terms with page references. Craft your `"system"` prompt to tell the chat API it should index the document and return the index entries and page references to you. Use the following `record` classes to create structured outputs as shown in Section 19.4.6:

Click here to view code image

```
public record IndexEntry(String entry, List<Integer> pages) {}
```

```
public record IndexEntries(List<IndexEntry> entries) {}
```

19.15 ***(Telephone-Number Word Generator)*** Enhance Exercise 11.3's telephone-number word generator to use genAI APIs to analyze the results by looking for actual words or combinations of shorter words that total seven letters (e.g., "PETSHOP" contains "PET" and "SHOP"). Use the OpenAI moderation API to remove offensive words and phrases.

19.16 ***(Building a PowerPoint Presentation)*** Create a genAI app that converts a document into a set of PowerPoint slides using the OpenAI chat and image-generation APIs. Allow the user to specify how detailed the PowerPoint slides should be and whether to use images from the source content, or generate appropriate images to embellish the slides.

19.17 ***(Text Tone Rewriter)*** Write a genAI app that receives a body of text with a negative tone provided by the user and prompts the OpenAI chat API to rewrite it in a neutral tone, then in a positive one. In each case, have the model explain its changes.

19.18 ***(Résumé Writer)*** Write a genAI app in which the user enters their qualifications and a job they're applying for, then prompts the OpenAI chat API to generate a polished brief résumé appropriate for that job.

19.19 ***(Summarizing Text)*** Write a genAI app that receives a document, then prompts the OpenAI chat API to create a summary abstract paragraph and a bullet list of key points.

19.20 ***(Document Explorer)*** Write a genAI app that receives a document then uses the OpenAI chat API to allow the user to converse with the document about its contents.

19.21 ***(Company Name Generator)*** Create a genAI app that allows the user to describe a company then prompts the OpenAI chat API to list potential company names, ensuring that each name is original.

19.22 ***(Query Reddit)*** Investigate the Reddit API. Create a genAI app that can retrieve the last five posts on a specific topic from Reddit. Use the OpenAI chat API to analyze the sentiment of each and the overall sentiment of all five posts.

19.23 ***(Chat with a Historical Figure)*** Create a genAI app that enables the user to chat with historical figures. Prompt the OpenAI chat API with a `"system"` message telling the model to act as if it's

the specified historical figure. Unlike ChatGPT's public interface, the chat API does not have "memory," so it does not remember the context of a conversation between calls. Investigate how to use `"assistant"` messages to pass the previous parts of the conversation as context for each new request.

Creative-Writing Apps

19.24 ***(Story Idea Generator)*** Write a genAI app that feeds text and images to the OpenAI chat API, then asks it to create ideas and an outline for an original book.

19.25 ***(Generate a Children's Story)*** Write a genAI app that receives an idea for a children's story then creates a polished story with meaningful images. Use the OpenAI moderation API to ensure there is no content inappropriate for children.

19.26 ***(Chat with a Book's Characters)*** Write a genAI app that receives a book as input then allows the user to chat with the book's characters. A great source of free books that are out of copyright in the United States is Project Gutenberg (`https://www.gutenberg.org`), which provides over 70,000 e-books in various formats, including plain text files. For information about Project Gutenberg's terms of use and copyright in other countries, see:

Click here to view code image

```
https://gutenberg.org/policy/terms_of_use.html
```

19.27 ***(Create a Poem)*** Write a genAI app that receives a poem idea and the name of a famous poet from the 1600s or 1700s. Use the OpenAI chat API to generate a poem in the style of the famous poet. Also, ask genAIs why they will not mimic the styles of recent poets.

19.28 ***(Generating Limericks)*** Use genAIs to learn about limericks. Create a genAI app that enables a user to submit a topic then uses the OpenAI chat API to generate a limerick based on that topic. Use the OpenAI moderation API to ensure the limerick does not contain inappropriate content. ChatGPT generated the following limerick when we asked for one about Java:[57]

57. *ChatGPT* in response to "Generate a funny limerick about Java programming." December 14, 2024. `https://chatgpt.com`.

Click here to view code image

```
There once was a Java buff, Lee,
```

Who wrestled with streams endlessly.
When the lambdas got wild,
They just sat back and smiled,
"At least it's not pointers in C!"

Text-to-Speech and Speech-to-Text Apps

19.29 ***(JavaFX Speech Translator with Playback)*** Convert Section 19.5.2's speech synthesis program into a JavaFX app. Enable the user to enter text and a language to translate it to, then synthesize speech in that language. Display the translated text and use JavaFX's media-playback capabilities to play the synthesized speech. Provide playback controls so the user can play and pause the audio.

19.30 ***(Advanced: JavaFX Speech Translator with Recording and Playback)*** Use genAIs to investigate the audio recording features of the `javax.sound.sampled` package. Enhance your solution to Exercise 19.29 with an option to record the user speaking, then transcribe the audio, display the transcription, translate the transcription to the language the user chooses and synthesize speech in that language.

19.31 ***(Speaking Limericks and Adding Music)*** Enhance your solution to Exercise 19.28 to speak the generated limericks. Prompt genAIs to determine how you can add music to your limericks.

19.32 ***(Google Cloud Text-to-Speech)*** Prompt genAIs for a simple tutorial on working with the Google Cloud Text-to-Speech APIs. Google offers a free tier for their basic voices (`https://cloud.google.com/text-to-speech/pricing`).

19.33 ***(Amazon Polly Text-to-Speech)*** Prompt genAIs for a simple tutorial on working with the Amazon Polly Text-to-Speech APIs. Amazon Polly offers free tiers for "neural voices" (their natural human-sounding AI voices) for 12 months after your first request.

19.34 ***(Text/Audio Chatbot)*** Build a text and audio chatbot that enables users to enter text or speak to the chatbot. For text, pass the text to OpenAI's chat API to get a response. For audio, transcribe the audio using Section 19.5.1's techniques, then send the transcription to OpenAI's chat API. The app should always display the chat response as text and provide an option to speak the response. When this option is selected, send the chat response to the text-to-speech API and play the response automatically.

19.35 ***(Advanced Project: Text/Audio Chatbot)*** This chapter's sample apps each waited for a response to complete, then displayed the result or saved it to a file. The OpenAI APIs support streaming responses, so apps can begin displaying text or playing audio responses immediately, rather than waiting for the complete response. Investigate OpenAI's streaming capabilities and SimpleOpenAI's streaming support, then enhance your solution to Exercise 19.34 to support streamed responses.

Spoken-Language Translation Apps

19.36 ***(United Nations Multi-Translator with Structured Output)*** Modify the example of Fig. 19.6 to translate the specified text into multiple languages. Using Fig. 19.6 as a guide, create a `"system"` message that asks the model to translate the text it receives to each of the specified languages. A `"user"` message should provide the text to translate. Ensure that your `"system"` message tells the model to return the data in structured format using the following record classes, which you should define in Section 19.9's `OpenAIUtilities` class:

Click here to view code image

```
public record Translation(String language, String translation) {}
public record Translations(String originalText,
   String originalLanguage, List<Translation> translations) {}
```

Call our `OpenAIUtilities` class's `chatWithStructuredOutput` method (Section 19.9.9) to perform the request and receive a `Translations` object containing the translations. Display the original source text and all the translations.

19.37 ***(United Nations Multi-Translator with Audio)*** Modify your Exercise 19.36 solution to uses the OpenAI speech-recognition and speech-synthesis APIs to add the ability to receive speech audio and speak the translations.

Code-Based Apps

19.38 ***(Generating Code from a Diagram)*** Create a genAI app that prompts the OpenAI chat API with a UML diagram or flowchart and requests the corresponding Java code.

19.39 ***(Generating Code from Pseudocode)*** Create a genAI app that prompts the OpenAI chat API with pseudocode and requests the corresponding Java code.

19.40 ***(Improving Code Performance)*** Create a genAI app that prompts the OpenAI chat API with the Java code from the die-rolling simulation in `RollDie.java` (located in the `resources` folder with this chapter's examples) and asks the model to evaluate the code for opportunities to improve its performance, reduce its memory use and parallelize it, then make the appropriate changes.

19.41 ***(Debugging)*** Create a genAI app that prompts the OpenAI chat API with the Java code from the broken die-rolling simulation in `BrokenRollDie.java` (located in the `resources` folder with this chapter's examples) and an explanation of what the code is supposed to do (roll 60 million dice and summarize their frequencies), then asks the chat API to locate and fix compilation errors and runtime logic errors.

19.42 ***(Refactoring)*** Create a genAI app that prompts the OpenAI chat API with the Java code from `Invoice.java` (located in the `resources` folder with this chapter's examples) and asks it to refactor the code from a `class` to a `record` that provides the same functionality.

19.43 ***(Improving Code Readability)*** Create a genAI app that prompts the OpenAI chat API with the uncommented Java code and poorly written Java code in `CrypticRoll- Die.java` (located in the `resources` folder with this chapter's examples) and asks the model to improve the code's readability by adding meaningful comments and renaming variables, methods and classes, as appropriate.

19.44 ***(Code Reviews)*** Create a genAI app that prompts the OpenAI chat API with Java code and asks it to check the code for style, code smells[58] and adherence to clean code guidelines,[59] SOLID principles[60,61] and the latest Java idioms.

58. "Code smell." Wikipedia. Wikimedia Foundation. October 13, 2024. https://en.wikipedia.org/wiki/Code_smell.

59. Martin, Robert C., *Clean Code: A Handbook of Agile Software Craftsmanship*. Upper Saddle River, NJ: Addison-Wesley, 2009.

60. Martin, Robert C., "Principles and Patterns," Object Mentor, archived September 6, 2015. Accessed February 25, 2025. https://web.archive.org/web/20150906155800/http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf.

61. Watts, Stephen, "SOLID Design Principles Explained: The 5 Class-Design Principles," BMC Blogs, August 26, 2024. Accessed February 25, 2025. https://www.bmc.com/blogs/solid-design-principles/.

19.45 ***(Translate Code Between Programming Languages)*** ChatGPT

can run Python code. Create a genAI app that prompts the OpenAI chat API with the Java code in the file `RollDieSmall.java` (located in the `resources` folder with this chapter's examples) and asks it to convert the code to Python then saves the Python code to a file. Test the resulting code by copying it into ChatGPT's web interface at `https://chatgpt.com` and asking it to run the code.

19.46 ***(Multimedia Tortoise and the Hare Race with JavaFX)*** In this exercise, you'll recreate the classic race of the tortoise and the hare. You'll use random-number generation to develop a simulation of this memorable event. Prompt genAIs to build a fun, animated JavaFX multimedia "extravaganza" in which the user can watch the race.

Our contenders begin the race at square 1 of 70 squares. Each square represents a possible position along the race course. The finish line is at square 70. The first contender to reach or pass square 70 is rewarded with a pail of fresh carrots and lettuce. The course weaves its way up a slippery mountain's side, so occasionally the contenders lose ground.

A clock ticks once per second. With each tick, your program should adjust the animal's positions according to the following rules:

Animal	Move	Percentage of the time
Tortoise	Fast plod	50%
	Slip	20%
	Slow plod	30%
Hare	Sleep	20%
	Fast hop	20%
	Big slip	10%
	Small hop	30%
	Small slip	20%

The program should keep track of the positions of the animals (i.e., position numbers are 1–70). Start each animal at position 1 (the "starting gate"). If an animal slips left before square 1, move it back to square 1.

Generate the percentages in the preceding table by producing a random integer i in the range $1 \leq i \leq 10$. For the tortoise, perform a "fast plod" when $1 \leq i \leq 5$, a "slip" when $6 \leq i \leq 7$ or a "slow plod" when $8 \leq i \leq 10$. Use a similar technique to move the hare.

For use in your race, we've provided in the `resources/race`

subfolder with this chapter's examples the following audio clips and images:[62]

62. Other than the public-domain William Tell Overture, we created these audios and images. If you prefer, you could search the web for others or create your own. Be sure to comply with the license terms for any media you'll use in your applications.

- An audio recording we created of the "Call to Post" trumpet piece played at the beginning of a horse race.
- A cannon-firing sound we created for our Cannon game.
- An audio clip we created of an announcer saying, "And they're off!" You could record yourself as the announcer saying this and other phrases to play throughout the race, such as "Tortoise pulls ahead!", "Hare pulls ahead!", "Down the stretch they come!", etc.
- A public-domain Wikimedia audio recording of the William Tell Overture[63], which we have edited down to just the gallop portion (bada bump, bada bump, bada bump bump bump…) and placed in this chapter's `resources` folder for you to play during the race.

  63. "File:Gioachino Rossini, William Tell Overture (military band version, 2000).ogg." Accessed February 25, 2025. `https://commons.wikimedia.org/wiki/File:Gioachino_Rossini,_William_Tell_Overture_(military_band_vers`

- Two slightly different tortoise and two slightly different hare images we created:

Tell the genAI to toggle between these images to create simple animations of the animals running. You can see the animations by viewing the `tortoise.gif` and `hare.gif` animated GIF images provided in the `resources/race` subfolder. You can use our images or ask the genAIs to create their own. Tell the genAIs that the app should perform the following tasks:

a. Before the race begins, **play the trumpet audio of the "Call to Post,"** signifying that the racers should take their mark. As the "Call to Post" plays, the tortoise and hare should appear from the screen's left side and take their positions.

b. **Play the cannon sound** to start the race, followed by the **announcer saying, "And they're off!"** At this point, the race animation begins.

c. Throughout the race, repeatedly play the provided William

Tell Overture's gallop portion in the background.

d. As the tortoise and the hare move across the screen, toggle between each animal's two images to make them appear to be running. The tortoise moves slower than the hare, so you may want to toggle between the tortoise's two images slower than between the hare's two images. When the hare arrogantly sleeps, stop toggling between its images.

e. When the tortoise and hare are in the same position, display "OUCH!" for the tortoise biting the hare and optionally play a high-pitched "OUCH!" audio clip.

f. If the tortoise wins, display "Tortoise wins!" and optionally play a "Tortoise wins!" audio clip followed by cheering. If the hare wins, display "Hare wins" and optionally play a "Hare wins" audio clip followed by booing. If the race ends in a tie, have the announcer say, "It's a tie!"

g. Play crowd cheering sounds, booing sounds and additional announcer commentary as appropriate throughout the race. Locate public-domain crowd sounds online, or ask genAIs to generate original sounds.

Image-Based Apps

19.47 ***(AI Art Critic)*** Create a genAI app in which the user uploads an art image or provides a URL of an online image. Use the OpenAI chat API's vision capabilities to analyze the image's artistic qualities and specify its genre.

19.48 ***(Creating New Fashion Designs)*** Create a JavaFX genAI app that uses the OpenAI image-generation API to allow the user to generate fashion designs based on text input. The app should display the generated images.

19.49 ***(Turning Fashion Sketches Into Images)*** Create a genAI app in which the user uploads a sketch. Use the OpenAI chat API to get a detailed description of the sketch, then send that description to the OpenAI image-generation API to generate a realistic image based on the sketch. Display the resulting image.

19.50 ***(Generating New Art in a Particular Style)*** Create a JavaFX genAI app that receives an art style and a description of a painting the user would like to create. Prompt the OpenAI image-generation API to generate an original image. Display the generated image.

19.51 ***(Steganography)*** Investigate steganography for hiding

messages or data in images, audio files or text. A common use of this is for "digital watermarking." Prompt genAIs, asking whether they use steganography to identify their generated content.

19.52 ***(2D Random Art)*** Create a JavaFX genAI app that creates random 2D art using JavaFX's 2D shape features. Use the OpenAI chat API to autogenerate the FXML for the shapes and a Java app that uses the FXML to create and display the art.

19.53 ***(2D Kinetic Art)*** Modify Exercise 19.52's app to animate the shapes, dynamically bouncing them off the edges of the JavaFX window and each other.

19.54 ***(3D Random Art)*** Create a JavaFX genAI app that creates random 3D art using JavaFX's 3D shape features. Use the OpenAI chat API to autogenerate the FXML for the shapes and a Java app that uses the FXML to create and display the art.

19.55 ***(3D Kinetic Art)*** Modify Exercise 19.54's app to animate the shapes, dynamically bouncing them off the edges of the JavaFX window and each other.

19.56 ***(Company Logo Generator)*** Create a JavaFX genAI app that enables the user to specify a company's name and specialty. Use the OpenAI image-generation API to generate potential logos for that company. Display each logo. Have other genAIs evaluate the logos and choose the best one.

19.57 ***(Convert Between UML Activity Diagrams and Flowcharts)*** Use genAIs to learn the basics of flowcharting, then create a JavaFX genAI app that can convert a UML activity diagram to a flowchart and vice versa. The app should

- use the OpenAI chat API's vision capabilities to analyze the original image and get a text representation of it,
- use the chat API to convert the text representation of the UML diagram to a text representation of a flowchart (or vice versa) then
- use the new text representation in a prompt to the OpenAI image-generation API, asking it to generate the new diagram.

19.58 ***(Convert from a UML Activity Diagram to Code)*** Create a JavaFX genAI app that generates code from a UML activity diagram. The app should use the OpenAI chat API's vision capabilities to analyze the UML diagram then generate Java code.

19.59 ***(Text Extraction Tool)*** Create a genAI app that uses the OpenAI chat API's vision capabilities to extract text from an image, such as a restaurant menu or a road sign.

19.60 ***(Draw a Floor Plan)*** Create a JavaFX genAI app that receives the dimensions of a room and a list of furniture and other items you'd like to place in it. Pass the details to the OpenAI chat API, asking it to create an enhanced description appropriate for use with the OpenAI image-generation API then send the description to the image-generation API to create a sample floor plan with user-specified items in place.

Audio, Speech and Music Apps

19.61 ***(Voice Cloning)*** Research the OpenAI API's Voice Engine for voice-cloning and why it is not available to the general public at

Click here to view code image

```
https://openai.com/index/navigating-the-challenges-and-
   opportunities-of-synthetic-voices/
```

19.62 ***(Search a Podcast)*** Create a JavaFX genAI app that receives a podcast audio file and a term or phrase to locate in the podcast. Use the OpenAI audio-transcription API to get a transcript with segments and their timestamps, as demonstrated in Section 19.7.1. Use the OpenAI chat API to search for the term or phrase in the transcript and determine the starting timestamp of the segment containing the term or phrase. Play the audio from a few seconds before the timestamp.

19.63 ***(Synthesize Speech in Multiple Voices from Text)*** Using Section 19.5.2's features, write a genAI app that synthesizes speech in all the voices listed at

Click here to view code image

```
https://platform.openai.com/docs/guides/text-to-speech
```

so the user can compare them. Which voice(s) do you prefer? Why?

19.64 ***(MIDI Music Generation)*** Prompt genAIs, asking them if they can create MIDI music. If so, ask them to generate original music in a style you specify. Play the result using QuickTime on macOS or the free, cross-platform VLC Media Player at

```
https://www.videolan.org/
```

Video

19.65 ***(OpenAI Sora)*** Check out OpenAI's Sora video demos at

```
https://sora.com
```

then experiment with it to create videos.

19.66 ***(Meta Movie Gen)*** Check out Meta's Movie Gen video demos at

Click here to view code image

```
https://ai.meta.com/research/movie-gen/
```

At the time of this writing, Movie Gen is not available. When it becomes accessible to the public, experiment with it to create videos.

19.67 ***(Google Veo 2)*** Investigate with Google Veo 2 video generator at

Click here to view code image

```
https://deepmind.google/technologies/veo/veo-2/
```

If it's accessible to the public, experiment with its features.

19.68 ***(Adobe Firefly)*** Check out Adobe's FireFly video demos at

```
https://firefly.adobe.com/
```

then experiment with it to create videos.

19.69 ***(Invideo AI)*** Check out Invideo AI's video demos at

```
https://invideo.io
```

then experiment with it to create videos.

19.70 ***(Which YouTube Video Should I Watch?)*** Write a genAI app that recommends the best videos for you to watch on a given subject based on the criteria you supply. Use the OpenAI chat API to specify the topic. Have it search YouTube and return video URLs.

Multimodal Apps

19.71 ***(Audio Summarization)*** Write a genAI app that combines transcription, summarization and speech synthesis. It should upload

an audio file, get its transcript, produce a summary abstract of the transcription, and synthesize speech from the summary abstract.

19.72 ***(Children's Book with Audio)*** Modify your solution to Exercise 19.25 to add audio narration.

Music-Based Apps

19.73 ***(Magenta AI Music Generation)*** Investigate Google TensorFlow's Magenta AI music generation capabilities at

Click here to view code image

```
https://magenta.tensorflow.org
```

and play with its Magenta demos at:

Click here to view code image

```
https://magenta.tensorflow.org/demos
```

19.74 ***(Magenta AI Music Generation)*** Prompt genAIs, asking how to use Magenta from a Java app. Build a JavaFX genAI app that creates and plays music in a user-specified style.

19.75 ***(Music Recommender System)*** Create a genAI app in which the user enters some favorite song titles and their artists, then use the OpenAI chat API to recommend other artists and songs the user might like. Use structured outputs (Section 19.4.6) for easier result processing.

Education-Based Apps

19.76 ***(Generate "What's Wrong with This Code?" Exercises)*** Write a genAI app that for a particular Java topic (e.g., arrays, `String`s, files) uses the OpenAI chat API to generate Java code snippets containing errors and asks the user to find the errors and specify the fixes. Use the OpenAI chat API to evaluate each fix and confirm whether it's correct.

19.77 ***(Generate "What Does This Code Do?" Exercises)*** Write a genAI app that uses the OpenAI chat API to generate, for a particular Java topic, code snippets and asks the user what the code does. Use the OpenAI chat API to evaluate the user's response and confirm whether it's correct.

19.78 ***(Java Coding Tutor)*** Create a genAI-powered Java coding

tutor that helps users learn Java. Your app should present topics to the user. To test the user's knowledge, provide fill-in-the-blank, multiple-choice and coding exercises of various difficulty levels. Other features could include:

- **Code Debugging Support**: Users paste in their code, and the app locates errors and suggests corrections.
- **Code Explanations**: Users paste in their code and ask a question about it, then the app explains it.

Use the OpenAI chat, speech recognition and speech synthesis APIs.

19.79 ***(Java Code-Solution Auto-Grader)*** Create a genAI app that enables an instructor to upload an exercise description, its Java code solution, students' solutions to the exercise and the instructor's grading criteria, then auto-grades the students' solutions.

19.80 ***(Quiz Builder)*** Create a genAI app that enables the user to provide a list of Java topics. Use the OpenAI chat API to build an interactive quiz with a user-specified number of multiple-choice questions. Use structured outputs (Section 19.4.6) for easier result processing.

19.81 ***(Generative AI: Computer-Assisted Instruction)*** Prompt genAIs to develop a command-line computer-assisted instruction program to help elementary school students improve their arithmetic skills. The program should prompt the student to select both a difficulty level and a type of arithmetic problem to study. The difficulty level determines the size of the numbers used:

- level 1 uses only single-digit numbers,
- level 2 includes up to two-digit numbers,
- and so on)

The problem type can be addition, subtraction, multiplication, division, or a random mix of all four.

The program should then generate a problem using a `RandomGenerator` object (or equivalent), display a question such as "How much is 6 times 7?" (adjusting the operation based on the selected type), and continue to ask the same question until the student provides the correct answer. When the student answers, the program should provide immediate feedback, selecting a response at random from a list of encouraging messages for a correct answer (e.g., "Very good!", "Excellent!", "Nice work!", "Keep up the good

work!”) or a list of prompts for an incorrect answer (e.g., “No. Please try again.”, “Wrong. Try once more.”, “Don't give up!”, “No. Keep trying.”).

The program should monitor the student's performance over a session of 10 questions, keeping count of correct and incorrect responses, and at the end, calculate the percentage of correct answers. Based on the results, if the student scores less than 75%, the program should display “Please ask your teacher for extra help.”, and if 75% or higher, it should congratulate the student with “Congratulations, you are ready to go to the next level!”. Finally, the program should reset itself so the user can begin a new session. Make the program multimodal, using the OpenAI chat, speech synthesis and speech recognition APIs.

19.82 ***(Law of Large Numbers with Coin Flipping: Static Visualization)*** Prompt genAIs to create a JavaFX app that demonstrates the Law of Large Numbers in action by flipping a coin 200 million times, displaying the results as a bar chart with one bar representing the number of heads and the other representing the number of tails.

19.83 ***(Law of Large Numbers with Coin Flipping: Dynamic Visualization)*** Prompt genAIs to modify Exercise 19.82 to display an animated bar chart. Allow users to specify how many coin flips to perform per animation frame. For example,

- when simulating only 200 coin flips, the user might want to see the bar chart update for every flip, but
- when simulating 200 million coin flips, the user might want to see the bar chart update for every 2000 flips.

19.84 ***(Parts-of-Speech Tagging with Structured Output)*** Modify Fig. 19.6 to perform parts-of-speech (POS) tagging—identifying each word’s part of speech, such as noun, verb, adjective, etc. Define the following `record` classes in class `OpenAIUtilities`:

Click here to view code image

```
public record PartOfSpeech(String text, String part) {}
public record PartsOfSpeech(List<PartOfSpeech> parts) {}
```

Using Fig. 19.6’s `"system"` message as a guide, create a `"system"` message that asks the model to analyze the text and return a structured JSON response with the parts of speech for the words in the `"user"` message. Ensure that your `"system"` message tells the

model to return only the data in the requested format and no additional information. Call our `OpenAIUtilities` class's `chatWithStructuredOutput` method (Section 19.9.9) to perform the request and get back a `PartsOfSpeech` object. Display the original source text and each word's part of speech.

Game Development

19.85 ***(SpotOn Game)*** The SpotOn game tests a user's reflexes by requiring the user to click moving spots before they disappear:

The required images and sounds for this exercise are located in the `resources` folder with this chapter's examples, though you can also use free images and sounds from many web-sites or create your own.

In this game, the spots shrink as they move, making them harder to click. The game begins on level one, and the user reaches each higher level by clicking ten spots. The higher the level, the faster the spots move—making the game increasingly challenging. When the user clicks a spot, the app plays a "hit" sound (`hit.mp3`), and the spot disappears. Points are awarded for each clicked spot—ten times the current level. Accuracy is important—any click that isn't on a spot plays a "miss" sound (`miss.mp3`) and decreases the score by 15 times the current level. You can tune the rewards and penalties to make the game more interesting.

The user begins the game with three additional "lives," which are displayed in the bottom-left corner of the app. If a spot disappears before the user clicks it, the app plays a "flushing" sound (`disappear.mp3`) sound, and the user loses a life. The user gains a life for reaching each new level, up to a maximum of seven lives. When no lives remain and a spot's animation ends before the spot is clicked, the game ends and displays an `Alert` dialog (package `javafx.scene.control`).

Prompt genAIs to generate the SpotOn game app using JavaFX. They should define a `Spot` subclass of `Circle` that encapsulates a spot's data and functionality. For each `Spot` the genAIs should:

a. Fill its circle with a spot image (`orange_spot.png` or `blue_spot.png`).

b. Provide random start and end points, and ensure that the `Spot` is completely within the `Pane`'s bounds.

c. Provide an event handler that responds to a mouse click within a `Spot`'s bounds, which you can set via the inherited

`Node` method `setOnMouseClicked`.

d. To make each `Spot` move, use a `ParallelTransition` consisting of a `Scale-Transition` that reduces the size of a spot from its original size to 25% of its original size and a `PathTransition` that moves the spot from its start point to its end point. Register an event handler that executes when the `ParallelTransition` runs to completion—in this case, the user should lose a life because the spot was not clicked.

The genAIs should use a `TimelineAnimation` to create the five initial spots—one every half-second. After that, each subsequent spot should be created when the user successfully clicks a spot or when a spot completes its animation and has not been clicked. To check for clicks that miss a spot, register a mouse-click handler for the app's `Pane`.

19.86 ***(Cannon Game)*** The Cannon game app challenges you to destroy nine targets before a ten-second time limit expires:

This game requires some two-dimensional analytic geometry and some basic trigonometry to determine the angle of the cannon. All of the required sounds for this exercise are located in the `resources` folder with this chapter's examples, though you also can use free sounds from reputable websites or create your own.

The game consists of a cannon, a cannonball, nine targets and a blocker that defends the targets—the blocker cannot be destroyed. The blocker and each target move vertically at different speeds, reversing direction when they hit the `Canvas`'s top or bottom. You aim and fire the cannon by clicking the mouse—the cannon's barrel then rotates to aim at the click point and fires the cannonball in a straight line in that direction.

You win by destroying all nine targets before time expires—if the timer reaches zero, you lose. When you destroy a target, the time remaining increases by a three-second time bonus. When you hit the blocker, the time remaining decreases by a three-second time penalty. When the game ends, the app displays an `Alert` dialog (package `javafx.scene.control`) indicating whether you won or lost, the number of shots fired and the elapsed time.

When you fire the cannon, the game plays a firing sound (`cannon_fire.wav`), and a cannonball begins moving from the cannon's barrel toward the blocker and targets. When a cannonball hits a target, a glass-breaking sound (`target_hit.wav`) plays and that target disappears. When the cannonball hits the blocker, a hit

sound (`blocker_hit.wav`) plays, and the cannonball bounces back toward the cannon.

Prompt genAIs to generate this game using a JavaFX `Circle`s and `Rectangle`s to represent the game elements and `TransitionAnimation`s to move them. Class `Shape` provides `static` method `intersect` that can be used for collision detection based on the actual shapes (that is, a `Circle` for the cannonball and `Rectangle`s for the blocker and targets). Tell the genAIs to consider defining classes that represent the various game elements.

19.87 ***(Word Search Puzzle Creator)*** Write a genAI JavaFX app that, given a list of words and the size of a word-search puzzle (e.g., 20-by-20 characters), forms an original word-search puzzle, similar to those shown at `https://thewordsearch.com/printable/`. The user should then be able to interact with the app to find the words.

19.88 ***(Word Search Solver)*** Write a JavaFX genAI app that receives an image of a word-search puzzle and a list of words to find. The app should use the OpenAI chat API's vision features to solve the puzzle, highlighting each word. Provide a mode in which the user can enter one word to find and the app locates the word, flashes it a few times, then permanently highlights it.

Assistant-Based Apps

19.89 ***(Résumé Filtering Assistant)*** Write a genAI app that receives a bunch of résumés and a list of items they should contain, then indicates which résumés meet these requirements.

19.90 ***(Research Assistant)*** Create a JavaFX genAI app that enables the user to specify a research topic. Use the OpenAI chat API to locate articles and videos about that topic, returning titles, URLs and brief summaries of each. Use structured outputs (Section 19.4.6) for easier result processing.

19.91 ***(Travel Planning Assistant)*** Create a genAI app that can assist a user with travel planning. The app should suggest sites to see at a particular destination and provide links to site videos.

Special Feature: Crossword Puzzles

19.92 ***(Challenging Project: Crossword Puzzle Solver)*** Create a genAI app that receives a photo of a crossword puzzle and its clues, then uses genAI to solve the puzzle. Provide a mode that allows the

user to specify one horizontal or vertical clue for which to get an answer.

19.93 ***(Challenging Project: Crossword Puzzle Generator)*** This is a complex problem. Given a topic, your app should use genAI APIs to generate a blank crossword puzzle, its clues and a solution. For example, when a company publishes its annual report and makes it available as a PDF online, your app could read the PDF and generate a puzzle of themed words and phrases from the report and flesh out the rest of the puzzle with related terms and phrases.

Capstone Project: The Multimodal GenAI Debating Society

19.94 ***(Challenging Project: GenAIs Debate One Another)*** Create a JavaFX genAI debate app in which two genAIs argue in favor of a topic (the pro side) and against a topic (the con side). A third genAI acts as the moderator, enforcing the debate rules and deciding the winner. The debate should follow the Oxford Style Debate rules at

Click here to view code image

```
https://www.uscourts.gov/about-federal-courts/educational-resources/
   about-educational-outreach/activity-resources/oxford
```

To make this project easier to implement, use a simplified version of the Oxford debate rules with no time restrictions. Rather than teams of three, each "team" will consist of one genAI. For simplicity, you could use just OpenAI's API, specifying each API call's role in the debate via `"system"` messages.

We prompted ChatGPT for the top high school and college debating topics of all time.[64] It suggested 40 topics. Here are several you might consider:

64. *ChatGPT* response to "Give me the top high school and college debating topics of all time." January 20, 2025. https://chatgpt.com.

- Does social media do more harm than good?
- Should the voting age be lowered to 16?
- Should college be free for all students?
- Should artificial intelligence be regulated?
- Should human cloning be allowed?

- Should cryptocurrencies replace traditional currencies?
- Should the U.S. Supreme Court have term limits?

Once the user chooses the topic, the debate should proceed as follows, using the chat and speech synthesis APIs:

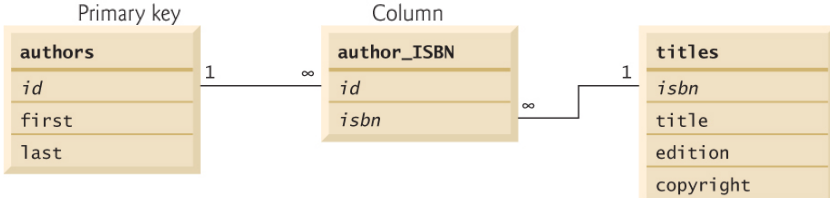
- The moderator says, “The topic of today’s debate is …” followed by the selected debate topic.
- The moderator tells the pro genAI to proceed.
- The pro genAI presents its opening argument, supporting the topic with logical points.
- The moderator tells the con genAI to proceed.
- The con genAI presents its opening argument with logical counterpoints.
- The moderator tells the pro genAI to present its rebuttal.
- The pro genAI responds to the arguments made by the con genAI, addressing specific points and countering them with logical reasoning and supporting evidence.
- The moderator tells the con genAI to present its rebuttal.
- The con genAI responds to the arguments made by the pro genAI, addressing specific points and countering them with logical reasoning and supporting evidence.
- The moderator invites the pro genAI to present a second round of rebuttals, addressing any new points raised by the con side.
- The pro genAI strengthens its argument by refuting the con side’s latest claims.
- The moderator invites the con genAI to present a second round of rebuttals, addressing any new points raised by the pro side.
- The con genAI strengthens its argument by refuting the pro side’s latest claims.
- The moderator genAI analyzes the debate, weighing the arguments and counterarguments.
- The moderator declares a winner based on the strength of reasoning, clarity and persuasiveness.

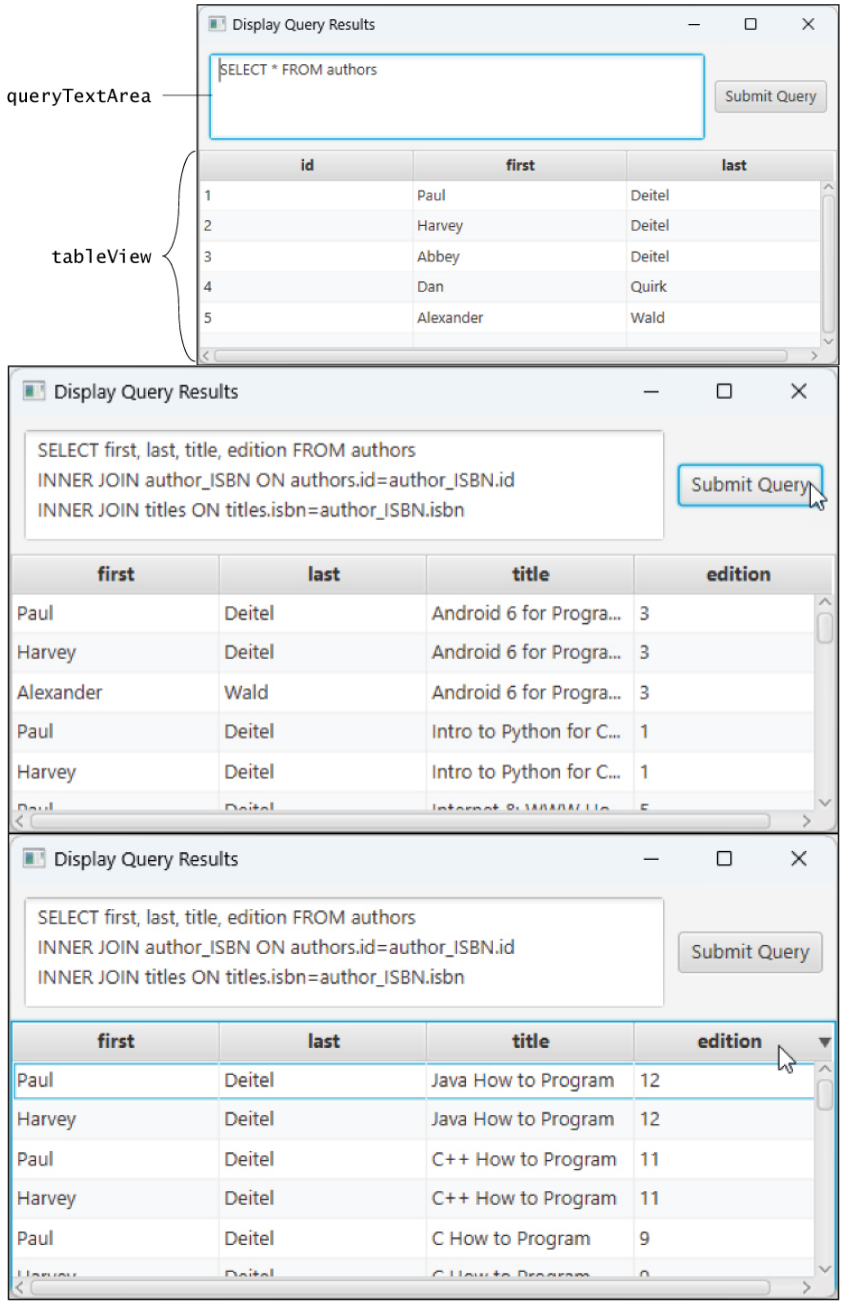
Number	Name	Department	Salary	Location
23603	Jones	413	1100	New Jersey
24568	Kerwin	413	2000	New Jersey
34589	Larson	642	1800	Los Angeles
35761	Myers	611	1400	Orlando
47132	Neumann	413	9000	New Jersey
78321	Stephens	611	8500	Orlando

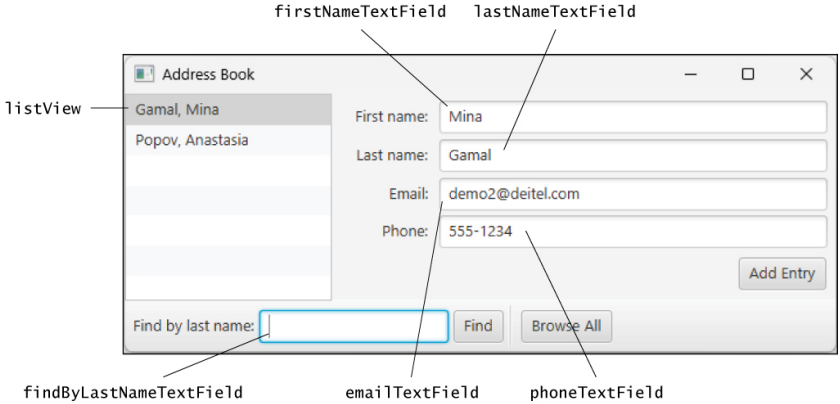
Row

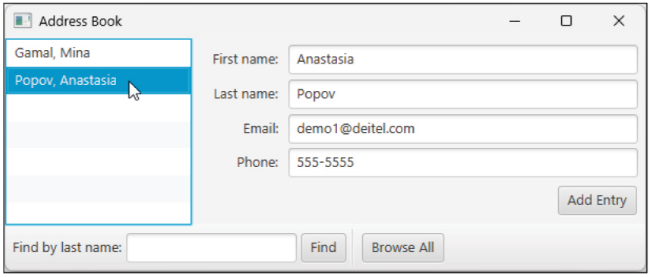
Primary key

Column

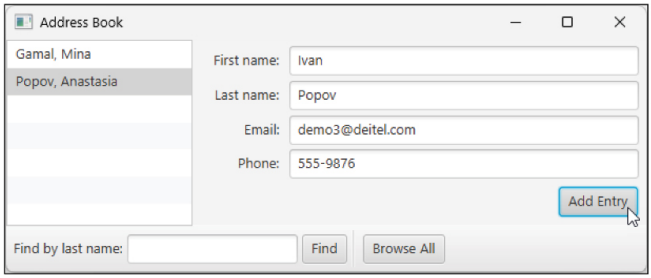



Display Query Results
queryTextArea
SELECT * FROM authors
Submit Query
tableView
id
first
last
1
Paul
Deitel
2
Harvey
Deitel
3
Abbey
Deitel
4
Dan
Quirk
5
Alexander
Wald
Display Query Results
SELECT first, last, title, edition FROM authors
INNER JOIN author_ISBN ON authors.id=author_ISBN.id
INNER JOIN titles ON titles.isbn=author_ISBN.isbn
Submit Query
first
last
title
edition
Paul
Deitel
Android 6 for Progra...
3
Harvey
Deitel
Android 6 for Progra...
3
Alexander
Wald
Android 6 for Progra...
3
Paul
Deitel
Intro to Python for C...
1
Harvey
Deitel
Intro to Python for C...
1
Display Query Results
SELECT first, last, title, edition FROM authors
INNER JOIN author_ISBN ON authors.id=author_ISBN.id
INNER JOIN titles ON titles.isbn=author_ISBN.isbn
Submit Query
first
last
title
edition
Paul
Deitel
Java How to Program
12
Harvey
Deitel
Java How to Program
12
Paul
Deitel
C++ How to Program
11
Harvey
Deitel
C++ How to Program
11
Paul
Deitel
C How to Program
9

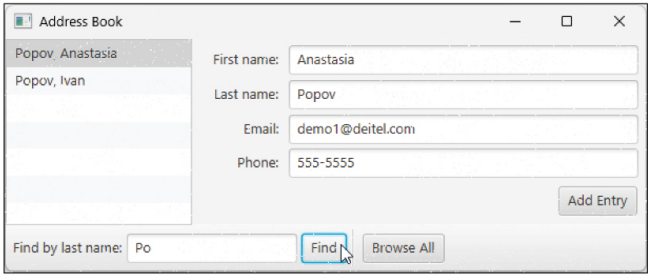

firstNameTextField
lastNameTextField
listView
Address Book
Gamal, Mina
Popov, Anastasia
First name:
Mina
Last name:
Gamal
Email:
demo2@deitel.com
Phone:
555-1234
Add Entry
Find by last name:
Find
Browse All
findByLastNameTextField
emailTextField
phoneTextField



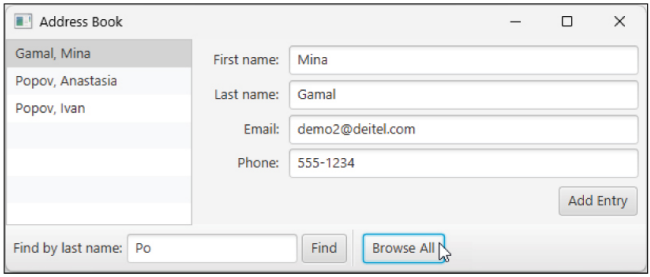
a) Viewing the entry for `Anastasia Popov`.



b) Adding a new entry for `Ivan Popov`.



c) Searching for last names that start with **Po**.



d) Returning to the complete list by selecting **Browse All**.

Accessing Databases with JDBC and SQLite

Objectives

In this chapter, you'll:

- Learn relational database concepts.
- Use Structured Query Language (SQL) and the SQLite command-line interface to retrieve data from and manipulate data in a SQLite database.
- Use the JDBC API's classes and interfaces to manipulate SQLite databases.
- Use JDBC's automatic JDBC driver discovery.
- Display query results in a JavaFX `TableView`.
- Bind query results to a `TableView` using JavaFX observable collections.
- Sort and filter a `TableView`'s contents.
- Create precompiled SQL statements with parameters via `PreparedStatement`s.
- Learn how transaction processing makes database applications more robust.

Outline

20.1 Introduction

20.2 Relational Databases

20.3 Setting Up the SQLite RDBMS

20.4 A `books` Database

20.4.1 `SELECT` Queries

20.4.2 `WHERE` Clause

20.4.3 `ORDER BY` Clause

20.4.4 Merging Data from Multiple Tables with `INNER JOIN`

20.4.5 `INSERT INTO` Statement
20.4.6 `UPDATE` Statement
20.4.7 `DELETE FROM` Statement
20.5 Connecting to and Querying a Database with JDBC
20.5.1 Automatic Driver Discovery
20.5.2 Connecting to the Database
20.5.3 Creating a `Statement` for Executing Queries
20.5.4 Executing a Query
20.5.5 Processing a Query's `ResultSet`
20.6 Querying the `books` Database
20.6.1 `DisplayQueryResults` App's GUI
20.6.2 `DisplayQueryResultsController` Class
20.7 `PreparedStatement`s
20.7.1 `AddressBook` App That Uses `PreparedStatement`s
20.7.2 `record` Class `Person`
20.7.3 Class `PersonQueries`
20.7.4 `AddressBook` GUI
20.7.5 Class `AddressBookController`
20.8 Stored Procedures
20.9 Transaction Processing
20.10 Wrap-Up
Special Section: Generative AI Exercises

20.1 Introduction

In Chapter 11, we demonstrated sequential text-file processing, working with plain-text, JSON-format and CSV-format data. These are useful when most or all of a file's data is to be processed. On the other hand, in transaction processing, it's crucial to locate and possibly update an individual data item quickly.

A **database** is an integrated collection of data. A **database**

management system (DBMS) provides mechanisms for storing and organizing data in a manner consistent with the database's format. Database management systems allow for convenient access and storage of data without concern for the internal representation of databases.

Relational Database Management Systems and Structured Query Language

Many of today's most popular database systems are **relational databases** (Section 20.2). **Relational database management systems** (**RDBMSs**) store data in **tables** and define relationships among the tables. **SQL (Structured Query Language)** is used almost universally with RDBMSs to manipulate data and perform **queries**, which request information that satisfies given criteria.[1]

1. The writing in this chapter assumes that SQL is pronounced as "see-quel." Some prefer "ess que el."

Popular Relational Database Management Systems

Popular open-source RDBMSs include SQLite, PostgreSQL, MariaDB and MySQL. These can be downloaded and used freely by anyone. All have support for Java. We'll use SQLite, which comes bundled with macOS and many Linux distributions and can be downloaded and installed on Windows. Some popular proprietary RDBMSs include Oracle, Microsoft SQL Server and IBM Db2.

JDBC

Java programs use the **JDBC (Java Database Connectivity) API** to interact with databases. A **JDBC driver** enables programs to connect to a database in a particular DBMS and manipulate that database using the JDBC API. The JDBC API is portable—the same code can manipulate databases in various RDBMSs.

Most popular database management systems provide JDBC drivers or have JDBC drivers provided by third parties. This chapter introduces JDBC and uses it to manipulate SQLite databases. The techniques we demonstrate here can be used to manipulate other databases with JDBC drivers. You can get the SQLite JDBC driver from

Click here to view code image

```
https://github.com/xerial/sqlite-jdbc/releases
```

For your convenience, we included this driver in the `libs` folder of this chapter's examples.

Generative AI

1 Prompt genAIs for the pros and cons of using databases vs. files for data storage in Java applications.

20.2 Relational Databases

A relational database is a logical table-based data representation that allows the data to be accessed without considering its physical structure. The following diagram contains a sample `Employee` table that might be used in a personnel system:

The table's primary purpose is to store employees' attributes. Tables are composed of **rows**, each describing a single entity. Here, each row represents one employee. Rows are composed of **columns** containing individual attribute values. The table above has six rows. The `Number` column represents the **primary key**—a column (or group of columns) with a unique value for each row. This guarantees that each row can be identified by its primary key. Examples of primary keys are social security numbers, employee ID numbers and part numbers in an inventory system—values in each are guaranteed to be unique. In this case, the rows are listed in ascending order by primary key, but they could be listed in descending order or no particular order at all.

Each column represents a different data attribute. Rows within a table are unique by primary key, but particular column values may be duplicated between rows. For example, three rows in the `Employee` table's `Department` column contain number 413.

Selecting Data Subsets

Database users are often interested in different data and different relationships among the data. Most users require only subsets of the rows and columns. Queries specify which subsets of the data to select from a table. You use Structured Query Language (SQL) to define queries. For example, you might select data from the `Employee` table to determine where each department is located,

presenting the data sorted in increasing order by department number, as in the following result. We'll discuss SQL shortly.

Click here to view code image

```
Department     Location

413            New Jersey
611            Orlando
642            Los Angeles
```

20.3 Setting Up the SQLite RDBMS

SQLite and its command-line tools come bundled with macOS and Linux. Windows users can download SQLite and its tools from

Click here to view code image

```
https://www.sqlite.org/download.html
```

Under the heading Precompiled Binaries for Windows, download SQLite[2] by selecting

2. Version `3470200` (for 3.47.2) was current at the time of this writing.

Click here to view code image

```
sqlite-tools-win-x64-3470200.zip
```

In your `Downloads` folder, right-click the ZIP file, select Extract All… then specify where to place the SQLite files. We placed the extracted files in our `Downloads` folder in the sub-folder named `sqlite-tools-win-x64-3470200`.

Add SQLite to the Windows Path Environment Variable

Next, add SQLite to the `Path` environment variable:

- Right-click the folder `sqlite-tools-win-x64-3470200` and select Copy as path.
- In the taskbar's Search field, enter `SystemPropertiesAdvanced`, and press or select *Enter*.

- In the System Properties dialog, select Environment Variables….
- Under System variables, select the `Path` variable, then select Edit….
- Select New, then paste in the path you copied in *Step 1* and select OK to save the changes.
- Close the Environment Variables and System Properties dialogs.

Generative AI

1 Though macOS and Linux bundle SQLite, the installed version may not be the latest one. Prompt genAIs for step-by-step instructions on how to upgrade SQLite on your system.

20.4 A `books` Database

This section presents a `books` database containing information about several of our books. We'll set up the database in SQLite using the file `books.sql` provided in the `ch20` examples folder. Then, we'll introduce the database's tables. We'll use this database via SQLite's command-line tools to introduce various database concepts, including operations that **create**, **read**, **update** and **delete** data—the so-called **CRUD operations**. As we introduce the tables, we'll execute SQL queries that output each table's contents.

Creating the `books` Database

In your command line, change to the `ch20` examples folder. Create the `books.db` database using the **`sqlite3` command** as follows:

- On macOS or in a Windows Command Prompt, enter the following command— the notation `<` provides the file `books.sql` as input to the `sqlite3` command:

```
sqlite3 books.db < books.sql
```

- In a Windows PowerShell, enter the following command to read `books.sql` then provide it as input to the `sqlite3`

command:

Click here to view code image

```
Get-Content books.sql | sqlite3.exe books.db
```

These commands do not display output at the command line if they successfully create the database.

Opening the `books` Database with the SQLite Command-Line Tools

Next, execute the following command to open the database in SQLite:

```
sqlite3 books.db
```

This launches the SQLite command-line tools and displays:

Click here to view code image

```
SQLite version 3.47.2 2024-12-07 20:39:59
Enter ".help" for usage hints.
sqlite>
```

As we introduce the database's tables, we'll execute commands here to list each table's contents. To list the data in a column-based format, execute the following SQLite command:

```
sqlite> .mode column
sqlite>
```

`authors` Table

The database has three tables—`authors`, `author_ISBN` and `titles`. The `authors` table stores all the authors and has three columns:

- `id`—The author's unique ID number. This integer column is defined as **autoincremented**—for each row inserted in the table, SQLite increases the `id` value by 1 to ensure that each row has a unique value. This column is the table's primary key.
- `first`—The author's first name (a string).
- `last`—The author's last name (a string).

Viewing the `authors` Table's Contents

Let's use a SQL query to view the `authors` table's contents:

Click here to view code image

```
sqlite> SELECT * FROM authors;
id  first      last
--  ---------  ------
1   Paul       Deitel
2   Harvey     Deitel
3   Abbey      Deitel
4   Dan        Quirk
5   Alexander  Wald
```

SQLite executes the query and outputs its results. A **`SELECT` query** gets rows and columns from one or more tables in a database. The **asterisk (`*`) wildcard** in the query selects all the columns from the `authors` table. We'll discuss `SELECT` queries in more detail shortly. Each SQL query you enter in the SQLite command-line tool must end with a semicolon (`;`). We show SQL keywords like `SELECT` and `FROM` in all capital letters by convention.

`titles` Table

The `titles` table stores all the books and has four columns:

- `isbn`—The book's ISBN (a string) is this table's primary key. ISBN is an abbreviation for "International Standard Book Number," a numbering scheme that publishers use to give every book a unique identification number.
- `title`—The book's title (a string).
- `edition`—The book's edition number (an integer).
- `copyright`—The book's copyright year (a string).

Use the following SQL query to list the `titles` table's contents:

Click here to view code image

```
sqlite> SELECT * FROM titles;
isbn        title                              edition  copyright
----------  ---------------------------------  -------  ---------
0135404673  Intro to Python for CS and DS      1        2020
0135224330  Python for Programmers              1        2019
0132151006  Internet & WWW How to Program       5        2012
0137598351  Java How to Program                 12       2025
0137398395  C How to Program                    9        2022
0133406954  Visual Basic 2012 How to Program   6        2014
0134601548  Visual C# How to Program           6        2017
```

```
0136151574  Visual C++ How to Program          2        2008
0136905692  C++20 for Programmers              2        2023
0138092427  C++ How to Program                 11       2024
0134444302  Android How to Program             3        2017
0134289366  Android 6 for Programmers          3        2016
```

`author_ISBN` Table

The `author_ISBN` table uses the following columns to associate authors from the `authors` table with their books in the `titles` table:

- `id`—An author's id (an integer).
- `isbn`—The book's ISBN (a string).

The `id` column is a **foreign key**, a column in this table that matches a primary-key column in another table—specifically, the `authors` table's `id` column. The `isbn` column is also a foreign key that matches the `titles` table's `isbn` primary-key column. A database might have many tables. A goal when designing a database is to minimize data duplication among the tables. To do this, each table represents a specific entity, and foreign keys help link the data in multiple tables. The primary keys and foreign keys are designated when you create the database tables (in our case, in the `books.sql` script).

Together, the `id` and `isbn` columns in this table form a **composite primary key**. Every row in this table uniquely matches one author to one book's ISBN. This table contains many entries, so let's use the SQL **`LIMIT` keyword** to restrict the results to the table's first five rows—`LIMIT` is not a standard SQL keyword, so it's not supported by all DBMSs:

Click here to view code image

```
sqlite> SELECT * FROM author_ISBN LIMIT 5;
id  isbn
--  ----------
1   0134289366
2   0134289366
5   0134289366
1   0135404673
2   0135404673
```

Every foreign-key value must appear as the primary-key value in a row of another table so the DBMS can ensure that the foreign-key

value is valid. This is known as the **Rule of Referential Integrity**. For example, the DBMS ensures that the `id` value for a particular `author_ISBN` row is valid by checking that there is a row in the `authors` table with that `id` as the primary key.

Foreign keys also allow related data in multiple tables to be selected from those tables and combined—this is known as **joining** the data. There is a **one-to-many relationship** between a primary key and a corresponding foreign key:

- one author can write many books, and
- one book can be written by many authors.

So, a foreign key can appear many times in its table but only once (as the primary key) in another table. For example, in the `books` database, the ISBN `0134289366` appears in several `author_ISBN` rows because this book has several authors, but it appears only once as a primary key in `titles`.

Entity-Relationship (ER) Diagram

The following **entity-relationship** (**ER**) **diagram** for the `books` database illustrates the database's tables and the relationships among them:
The first compartment in each box contains the table's name, and the remaining compartments contain the table's columns. The names in italics are primary keys. A table's primary key uniquely identifies each row in the table. Every row must have a primary-key value, and that value must be unique in the table. This is known as the **Rule of Entity Integrity**. Again, for the `author_ISBN` table, the primary key is the combination of both columns— this is known as a composite primary key.

The lines connecting the tables represent the relationships among the tables. Consider the line between `authors` and `author_ISBN`. There's a `1` on the `authors` end and an infinity symbol (∞)on the `author_ISBN` end. This indicates a one-to-many relationship. For each author in the authors table, there can be an arbitrary number of ISBNs for books written by that author in the `author_ISBN` table—that is, an author can write any number of books, so an author's `id` can appear in multiple rows of the `author_ISBN` table. The relationship line links the `id` column in the `authors` table (where `id` is the primary key) to the `id` column in the `author_ISBN` table (where `id` is a foreign key). The line between the tables links the primary key to the matching foreign

key.

The line between the `titles` and `author_ISBN` tables illustrates a one-to-many relationship—one book can be written by many authors. The line links the primary key `isbn` in table `titles` to the corresponding foreign key in table `author_ISBN`. The relationships in the entity-relationship diagram illustrate that the sole purpose of the `author_ISBN` table is to provide a **many-to-many relationship** between the `authors` and `titles` tables—an author can write many books, and a book can have many authors.

SQL Keywords

The following subsections continue our SQL presentation in the context of our `books` database, demonstrating SQL queries and statements using the SQL keywords in the following table. Other SQL keywords are beyond this text's scope:

SQL keyword	Description
`SELECT`	Retrieves data from one or more tables.
`FROM`	Tables involved in the query. Required in every `SELECT`.
`WHERE`	Criteria for selection that determine the rows to be retrieved, deleted or updated. Optional in a SQL statement.
`GROUP BY`	Criteria for grouping rows. Optional in a `SELECT` query.
`ORDER BY`	Criteria for ordering rows. Optional in a `SELECT` query.
`LIMIT`	Limit the number of results returned in a `SELECT` query. `LIMIT` is not a standard SQL keyword, so it's not supported by all DBMSs.
`INNER JOIN`	Merge rows from multiple tables.
`INSERT`	Insert rows into a specified table.
`UPDATE`	Update rows in a specified table. **Always combine this with a `WHERE` clause specifying which rows to update; otherwise, all rows will be updated.**
`DELETE`	Delete rows from a specified table. **Always combine this with a `WHERE` clause specifying which rows to delete; otherwise, all rows will be deleted.**

20.4.1 `SELECT` Queries

The previous section used SELECT statements and the `*` wildcard character to get all the columns from a table. Typically, you need only a subset of the columns. To retrieve only the specific columns you need, specify a comma-separated list of column names. For example, let's retrieve only the columns `first` and `last` from the `authors` table:

Click here to view code image

```
sqlite> SELECT first, last FROM authors;
first      last
---------  ------
Paul       Deitel
Harvey     Deitel
Abbey      Deitel
Dan        Quirk
Alexander  Wald
```

20.4.2 `WHERE` Clause

You'll often select rows in a database that satisfy specific **selection criteria**, especially in big data, where a database might contain millions or billions of rows. Only rows meeting the selection criteria are selected. SQL's **`WHERE` clause** specifies a query's selection criteria. Let's select the `title`, `edition` and `copyright` for all books with copyright years greater than 2019. String values in SQL queries are delimited by single (`'`) quotes, as in `'2019'`:

Click here to view code image

```
sqlite> SELECT title, edition, copyright
   ...> FROM titles
   ...> WHERE copyright > '2019';
title                          edition   copyright
-----------------------------  -------   ---------
Intro to Python for CS and DS  1         2020
Java How to Program            12        2025
C How to Program               9         2022
C++20 for Programmers          2         2023
C++ How to Program             11        2024
```

You can enter lengthy SQL queries over several lines for readability. SQLite will not execute a query until you terminate it with a semicolon and press (or select) *Enter*. Before each line of the query after the first, SQLite displays the continuation prompt `"...>"` to indicate that the query is not complete.

Pattern Matching: Zero or More Characters

The `WHERE` clause may contain the operators `<`, `>`, `<=`, `>=`, `=`, `<>` (not equal) and `LIKE`. Operator **`LIKE`** is used for **pattern matching**—searching for strings that match a given pattern. A pattern containing the **percent (`%`) wildcard character** searches for strings

with zero or more characters at the percent character's position. For example, let's locate all authors whose last name starts with the letter `D`:

Click here to view code image

```
sqlite> SELECT id, first, last
   ...> FROM authors
   ...> WHERE last LIKE 'D%';
id  first   last
--  ------  ------
1   Paul    Deitel
2   Harvey  Deitel
3   Abbey   Deitel
```

Pattern Matching: Any Character

An **underscore** (_) in the pattern string indicates a single wildcard character at that position. Let's select the rows of all the authors whose last names start with any character, followed by the letter `b`, followed by any number of additional characters (specified by `%`):

Click here to view code image

```
sqlite> SELECT id, first, last
   ...> FROM authors
   ...> WHERE first LIKE '_b%';
id  first  last
--  -----  ------
3   Abbey  Deitel
```

20.4.3 `ORDER BY` Clause

The **`ORDER BY` clause** sorts a query's results into ascending order (lowest to highest) or descending order (highest to lowest), specified with `ASC` and `DESC`, respectively. The default sorting order is ascending, so `ASC` is optional. Let's sort the titles in ascending order:

Click here to view code image

```
sqlite> SELECT title FROM titles ORDER BY title ASC;
title
--------------------------------
Android 6 for Programmers
Android How to Program
C How to Program
C++ How to Program
```

```
C++20 for Programmers
Internet & WWW How to Program
Intro to Python for CS and DS
Java How to Program
Python for Programmers
Visual Basic 2012 How to Program
Visual C# How to Program
Visual C++ How to Program
```

Sorting By Multiple Columns

To sort by multiple columns, use a comma-separated list of column names after `ORDER BY`. Let's sort author names by `last`, then `first` for any authors who have the same last name:

Click here to view code image

```
sqlite> SELECT id, first, last
   ...> FROM authors
   ...> ORDER BY last, first;
id  first      last
--  ---------  ------
3   Abbey      Deitel
2   Harvey     Deitel
1   Paul       Deitel
4   Dan        Quirk
5   Alexander  Wald
```

The sorting order can vary by column. Let's sort the authors in descending order by last name and ascending order by first name for any authors who have the same last name:

Click here to view code image

```
sqlite> SELECT id, first, last
   ...> FROM authors
   ...> ORDER BY last DESC, first ASC;
id  first      last
--  ---------  ------
5   Alexander  Wald
4   Dan        Quirk
3   Abbey      Deitel
2   Harvey     Deitel
1   Paul       Deitel
```

Combining the `WHERE` and `ORDER BY` Clauses

The `WHERE` and `ORDER BY` clauses can be combined in one query. Let's get the `isbn`, `title`, `edition` and `copyright` of each book in

the `titles` table that has a `title` ending with `'How to Program'` and sort them in ascending order by `title`.

Click here to view code image

```
sqlite> SELECT isbn, title, edition, copyright
   ...> FROM titles
   ...> WHERE title LIKE '%How to Program'
   ...> ORDER BY title;
isbn        title                               edition  copyright
----------  ----------------------------------  -------  ---------
0134444302  Android How to Program               3        2017
0137398395  C How to Program                     9        2022
0138092427  C++ How to Program                   11       2024
0132151006  Internet & WWW How to Program        5        2012
0137598351  Java How to Program                  12       2025
0133406954  Visual Basic 2012 How to Program     6        2014
0134601548  Visual C# How to Program             6        2017
0136151574  Visual C++ How to Program            2        2008
```

20.4.4 Merging Data from Multiple Tables with `INNER JOIN`

The `books` database's `author_ISBN` table links authors to their corresponding titles. If we did not separate this information into individual tables, we'd need to include author information with each entry in the `titles` table, resulting in duplicate information for authors who wrote multiple books. You can merge data from multiple tables, referred to as joining the tables, with **`INNER JOIN`**. Let's produce a list of authors accompanied by their books' ISBNs. There are many results for this query, so we limited the results to the first five rows:

Click here to view code image

```
sqlite> SELECT first, last, isbn
   ...> FROM authors
   ...> INNER JOIN author_ISBN ON authors.id = author_ISBN.id
   ...> ORDER BY last, first LIMIT 5;
first   last    isbn
------  ------  ----------
Abbey   Deitel  0132151006
Abbey   Deitel  0133406954
Harvey  Deitel  0134289366
Harvey  Deitel  0135404673
Harvey  Deitel  0132151006
```

The `INNER JOIN`'s **`ON` clause** uses a primary-key column in one table and a foreign-key column in the other to determine which rows to merge from each table. This query merges the `authors` table's `first` and `last` columns with the `author_ISBN` table's `isbn` column, sorts the results in ascending order by `last` then `first`, and selects only the first five results.

The syntax `authors.id` *(table_name.column_name)* in the `ON` clause uses **qualified name** syntax—this is required if the columns have the same name in both tables. This syntax can be used in any SQL statement to distinguish columns in different tables that have the same name. In some systems, table names qualified with the database name can be used to perform cross-database queries. As always, the query can contain an `ORDER BY` clause.

20.4.5 `INSERT INTO` Statement

To this point, you've queried existing data. Sometimes, you'll execute SQL statements that modify the database. The **`INSERT INTO` statement** inserts a row into a table. Let's insert a new author named `Mei-ling Liu` into the `authors` table:

Click here to view code image

```
sqlite> INSERT INTO authors (first, last)
   ...> VALUES ('Mei-ling', 'Liu');
```

The SQL keywords `INSERT INTO` are followed by the table in which to insert the new row and a comma-separated list of column names in parentheses. The list of column names is followed by the **SQL keyword `VALUES`** and a comma-separated list of values in parentheses. The values provided must match the column names specified in order and type.

We do not specify a value for the `id` column because it's an autoincremented column in the `authors` table—this was defined in the script `books.sql` that created the table. For every new row, SQLite assigns a unique `id` value that is the next value in the autoincremented sequence (i.e., 1, 2, 3, …). In this case, `Mei-ling Liu` is assigned `id` number `6`. To confirm this, let's query the `authors` table's contents:

Click here to view code image

```
sqlite> SELECT id, first, last FROM authors;
id  first      last
--  ---------  ------
```

```
1   Paul        Deitel
2   Harvey      Deitel
3   Abbey       Deitel
4   Dan         Quirk
5   Alexander   Wald
6   Mei-ling    Liu
```

Note Regarding Strings That Contain Single Quotes

SQL delimits strings with single quotes (`'`). A string containing a single quote, such as O'Malley, must have two single quotes in the position where the single quote appears (e.g., `'O''Malley'`). The first acts as an escape character for the second. Not escaping single-quote characters in a string that's part of a SQL statement is a SQL syntax error.

20.4.6 `UPDATE` Statement

An **`UPDATE` statement** modifies existing values. Let's assume that Mei-ling Liu's last name is incorrect in the database and update it to `'Cai'`:

Click here to view code image

```
sqlite> UPDATE authors SET last='Cai'
   ...> WHERE last='Liu' AND first='Mei-ling';
```

The `UPDATE` keyword is followed by the table to update, the **keyword `SET`** and a comma-separated list of *column_name* `=` *value* pairs indicating the columns to change and their new values. The change will be applied to every row if you do not specify a `WHERE` clause. The `WHERE` clause in this query indicates that we should update only rows with the last name `'Liu'` and the first name `'Mei-ling'`.

Of course, multiple people could have the same first and last name. To make a change to only one row, use the row's unique primary key in the `WHERE` clause. In this case, we could have specified:

```
WHERE id = 6
```

For statements that modify the database, you can use the **SQLite `changes` function** to check the number of rows that were modified. The following confirms that the `UPDATE` modified one row:

```
sqlite> SELECT changes();
changes()
---------
1
```

We also can confirm the update by listing the `authors` table's contents:

Click here to view code image

```
sqlite> SELECT id, first, last FROM authors;
id  first       last
--  ----------  ------
1   Paul        Deitel
2   Harvey      Deitel
3   Abbey       Deitel
4   Dan         Quirk
5   Alexander   Wald
6   Mei-ling    Cai
```

20.4.7 `DELETE FROM` Statement

A **`DELETE FROM` statement** removes rows from a table. Let's remove Mei-ling Cai from the `authors` table using her author ID:

Click here to view code image

```
sqlite> DELETE FROM authors WHERE id=6;
```

The optional `WHERE` clause determines which row(s) to delete. If `WHERE` is omitted, all the table's rows are deleted. Here's the `authors` table after the `DELETE` operation:

Click here to view code image

```
sqlite> SELECT id, first, last FROM authors;
id  first       last
--  ----------  ------
1   Paul        Deitel
2   Harvey      Deitel
3   Abbey       Deitel
4   Dan         Quirk
5   Alexander   Wald
```

Closing the Database and Exiting SQLite

When you're done interacting with the database at the command line, execute the following command to close the database and exit

the SQLite command-line tools.

```
sqlite> .exit
```

Generative AI

1 Prompt genAIs with the `books.sql` script, asking for a tutorial on the SQL `WHERE` clause's options in `SELECT` queries with examples in the context of the `books` database and SQLite. Use the SQLite command-line tools to try the suggested SQL queries.

2 Prompt genAIs with the `books.sql` script, asking them to write an `INNER JOIN` query that matches the author Paul Deitel with the ISBNs and titles for which he's an author. Also, ask the genAIs to explain the query step-by-step. Execute the query using the SQLite command-line tools to confirm the results.

3 Prompt genAIs for a list of the most commonly used SQLite functions, like `changes`, for interacting with a database using SQLite's command-line tools.

4 Prompt genAIs for a list of the most commonly used SQLite commands, like `.mode` and `.exit`, for interacting with SQLite.

20.5 Connecting to and Querying a Database with JDBC

The example of Fig. 20.1 performs a simple query on the `books` database that retrieves the entire `authors` table and displays the data. The program illustrates connecting to the database, querying the database and processing the result. We broke the program into parts for discussion purposes. After compiling this program, you can run it with the following command, which adds the SQLite driver to the classpath—Windows users should replace the colon (`:`) with a semicolon (`;`) and the forward slashes (`/`) with backslashes (`\`):

Click here to view code image

```
java -cp ".:../libs/sqlite-jdbc-3.47.1.0.jar" DisplayAuthors
```

In this chapter's code, we assume the book's `examples` folder is located in your user account's `Documents` folder, as discussed in the book's Before You Begin section. If not, you must update

lines 14–15 and similar lines in subsequent examples.

Click here to view code image

```
1  // Fig. 20.1: DisplayAuthors.java
2  // Displaying the contents of the authors table.
3  import java.nio.file.Path;
4  import java.sql.Connection;
5  import java.sql.Statement;
6  import java.sql.DriverManager;
7  import java.sql.ResultSet;
8  import java.sql.ResultSetMetaData;
9  import java.sql.SQLException;
10
11 public class DisplayAuthors {
12    public static void main(String args[]) {
13       // books.db location
14       Path dbPath = Path.of(System.getProperty("user.home"),
15          "Documents", "examples", "ch20", "books.db");
16
17       final String url = "jdbc:sqlite:" + dbPath.toUri();
18       final String query = "SELECT id, first, last FROM authors";
19
20       // use try-with-resources to connect to and query the database
21       try (
22          Connection connection = DriverManager.getConnection(url);
23          Statement statement = connection.createStatement();
24          ResultSet resultSet = statement.executeQuery(query)) {
25
26          // get ResultSet's meta data
27          ResultSetMetaData metaData = resultSet.getMetaData();
28          int numberOfColumns = metaData.getColumnCount();
29
30          System.out.printf("Authors Table of Books Database:%n%n");
31
32          // display the names of the columns in the ResultSet
33          for (int i = 1; i <= numberOfColumns; i++) {
34             System.out.printf("%-11s", metaData.getColumnName(i));
35          }
36          System.out.println();
37
38          // display query results
39          while (resultSet.next()) {
40             for (int i = 1; i <= numberOfColumns; i++) {
41                System.out.printf("%-11s", resultSet.getObject(i));
42             }
43             System.out.println();
44          }
45       }
46       catch (SQLException sqlException) {
```

```
47              sqlException.printStackTrace();
48          }
49      }
50  }
```

```
Authors Table of Books Database:

id          first       last
1           Paul        Deitel
2           Harvey      Deitel
3           Abbey       Deitel
4           Dan         Quirk
5           Alexander   Wald
```

Fig. 20.1 | Displaying the contents of the `Authors` table.

Lines 4–9 import the JDBC interfaces and classes from package `java.sql` used in this program. The `main` method connects to the `books` database, queries the database, displays the query result and closes the database connection. Line 17 declares a `String` constant for the database URL (also known as a connection string). This identifies the database to connect to and information about the protocol used by the JDBC driver (discussed shortly). Line 18 declares a `String` constant representing the SQL query that will select the `id`, `first` and `last` columns from the database's `authors` table.

20.5.1 Automatic Driver Discovery

JDBC supports **automatic driver discovery**—it loads the database driver into memory for you. To ensure that the program can locate the driver class, you must include the class's location in the program's classpath[3] when you execute the program, as shown in the `java` command above Fig. 20.1.

3. In a modularized application (Chapter 21, Java Platform Module System), you'd include the driver's module on the module path.

20.5.2 Connecting to the Database

The JDBC interfaces we use in this example extend the `AutoCloseable` interface, so you can use objects that implement these interfaces with the `try`-with-resources statement (lines 21–48). Lines 22–24 create this example's `AutoCloseable` objects, which are closed when the `try` block terminates (line 45) or if an

exception occurs during the `try` block's execution. Each object created in the parentheses following keyword `try` must be separated from the next by a semicolon (`;`).

Line 22 creates a **`Connection` object** that manages the connection between the Java program and the database. `Connection` objects enable programs to create SQL statements that manipulate databases. The program initializes `connection` with the result of a call to the **`DriverManager` class's `static` method `getConnection`**. The method's argument is a `String` specifying the database URL. This method attempts to connect to the database specified by its URL argument—if it cannot, it throws a **`SQLException`**.

The URL locates the database. In this chapter's examples, the database is a file on the local computer in this chapter's examples folder, but it could reside on a network. The JDBC URL begins with `"jdbc:sqlite:"`, specifying:

- the **protocol** for communication (`jdbc`) and
- the **subprotocol** for communication (`sqlite`).

This is followed by the SQLite database file's location (created in lines 14–15).

The subprotocol `sqlite` indicates that the program uses a SQLite-specific subprotocol to connect to the database. The following table lists the JDBC driver names and database URL formats of several popular RDBMSs:

RDBMS	URL format
MySQL	`jdbc:mysql://`*hostname*`:`*portNumber*`/`*databaseName*
Oracle	`jdbc:oracle:thin:@`*hostname*`:`*portNumber*`/`*databaseName*
IBM DB2	`jdbc:db2://`*hostname*`:`*portNumber*`/`*databaseName*
PostgreSQL	`jdbc:postgresql://`*hostname*`:`*portNumber*`/`*databaseName*
Microsoft SQL Server	`jdbc:sqlserver://`*hostname*`:`*portNumber*`;databaseName=`*databaseName*
MariaDB	`jdbc:mariadb://`*hostname*`:`*portNumber*`/`*databaseName*

Most database management systems require users to log in before accessing the database contents. `DriverManager` method `getConnection` is overloaded with versions that enable the program to supply the username and password to gain access.

20.5.3 Creating a `Statement` for Executing Queries

Line 23 of Fig. 20.1 invokes **`Connection` method `createStatement`** to obtain an object that implements interface `Statement` (package `java.sql`). You use a **`Statement` object** to submit SQL statements to the database.

20.5.4 Executing a Query

Line 24 uses the **`Statement` object's `executeQuery` method** to submit a query that selects all the author information from the `Authors` table. This method returns an object that implements the **`ResultSet` interface** and contains the query results.

20.5.5 Processing a Query's `ResultSet`

Lines 27–44 process the `ResultSet`.

Displaying the Column Names

Line 27 obtains the **`ResultSet`'s `ResultSetMetaData` object** (package `java.sql`). The **metadata** describes the `ResultSet`'s contents. Programs can use metadata programmatically to get information about the `ResultSet`'s column names and types. Line 28 uses **`ResultSetMetaData` method `getColumnCount`** to retrieve the number of columns in the `ResultSet`. Lines 33–35 display the column names. Metadata enables programs to process `ResultSet` contents dynamically when detailed information about the `ResultSet` is not known in advance.

Displaying Each Row's Data

Lines 39–44 display the data in each `ResultSet` row. First, the program positions the **`ResultSet` cursor** (which points to the row being processed) to the first row. **Method `next`** (line 39) returns the `boolean` value `true` if it can position to the next row; otherwise, the method returns `false` to indicate there are no more results. Initially, a `ResultSet` cursor is positioned before the first row. A `SQLException` occurs if you access a `ResultSet`'s contents before positioning the `ResultSet` cursor to the first row with method `next`.

If the `ResultSet` has rows, lines 40–42 extract and display the

current row's columns. Each column can be extracted as a specific Java type—the `ResultSetMetaData` **method** `getColumnType` returns a constant from **class** `Types` (package `java.sql`) indicating a given column's type. Programs can use these values in a `switch` statement to invoke `ResultSet` methods that return the column values as appropriate Java types. For example, if the column's type is `Types.INTEGER`, `ResultSet` **method** `getInt` gets the column value as an `int`. For simplicity, this example treats each value as an `Object`. We retrieve each column value with `ResultSet` **method** `getObject` (line 41), then display the `Object`'s `String` representation. `ResultSet` *get* methods typically receive as an argument either a column number (as an `int`) or a column name (as a `String`) indicating which column's value to obtain. **Unlike array indices, `ResultSet` column numbers start at 1.** Specifying column index 0 when obtaining values from a `ResultSet` causes a `SQLException`.

Accessing Columns By Column Number vs. Column Name

If a query specifies the exact columns to select from the database, the `ResultSet` contains the columns in the specified order. For this scenario, using the column number to obtain the column's value is more efficient than using the column name. The column number provides direct access to the specified column. Using the column name requires a search of the column names to locate the appropriate column. However, using column names to obtain values from a `ResultSet` is less error-prone than obtaining values by column number—you don't need to remember the column order. Also, your code does not have to change if the column order changes.

Closing the Database Resources

When line 45 is reached, the `ResultSet`, `Statement` and `Connection` objects' `close` methods are called to close the corresponding resources. If you attempt to manipulate a `ResultSet` after closing the `Statement` that created it, a `SQLException` occurs—the `ResultSet` is discarded when the `Statement` is closed.

Each `Statement` object can open only one `ResultSet` object at a time. When a `Statement` returns a new `ResultSet`, the `Statement` closes the prior `ResultSet`. To use multiple `ResultSet`s in parallel, separate `Statement` objects must return the `ResultSet`s.

Generative AI

1 Prompt genAIs to overview the most commonly used JDBC `ResultSetMetaData` methods.

2 Prompt genAIs to overview the most commonly used JDBC `ResultSet` methods.

20.6 Querying the `books` Database

Next, we present a `DisplayQueryResults` app that allows you to enter and execute queries. We use an `ObservableList` and data binding to present the query results in a JavaFX `TableView`. We populate the `ObservableList` dynamically from a `ResultSet`.

20.6.1 `DisplayQueryResults` App's GUI

The following diagram contains the app's GUI (defined in `DisplayQueryResults.fxml`) labeled with its fx:ids:
You've built many FXML GUIs in prior chapters, so we point out only the key elements and the event-handler methods implemented in class `DisplayQueryResultsController` (Fig. 20.2). For the complete layout details, open the file `DisplayQueryResults.fxml` in Scene Builder or view the FXML in a text editor. The GUI's primary layout is a `BorderPane`:

- The top area is a `GridPane` containing a `TextArea` and a `Button` for entering and submitting a query.
- The center area contains a `TableView` to display the query results.

The controller class defines the event-handling method `submitQueryButtonPressed`, which is called when the user presses or selects the Submit Query `Button`.

Compiling the App

You can compile the `DisplayQueryResults` app with the following command on macOS:

Click here to view code image

```
javac --module-path $PATH_TO_FX \
   --add-modules javafx.controls,javafx.fxml *.java
```

On Windows, use the following command in PowerShell:

Click here to view code image

```
javac --module-path $env:PATH_TO_FX `
   --add-modules javafx.controls,javafx.fxml *.java
```

or following command in a Command Prompt:

Click here to view code image

```
javac --module-path %PATH_TO_FX% ^
   --add-modules javafx.controls,javafx.fxml *.java
```

Running the App

To run the `DisplayQueryResults` app, use the following command on macOS:

Click here to view code image

```
java --module-path $PATH_TO_FX \
   --add-modules javafx.controls,javafx.fxml \
   -cp ".:../libs/sqlite-jdbc-3.47.1.0.jar" DisplayQueryResults
```

On Windows, use the following command in PowerShell:

Click here to view code image

```
java --module-path $env:PATH_TO_FX `
   --add-modules javafx.controls,javafx.fxml `
   -cp ".;..\libs\sqlite-jdbc-3.47.1.0.jar" DisplayQueryResults
```

or following command in a Command Prompt:

Click here to view code image

```
java --module-path %PATH_TO_FX% ^
   --add-modules javafx.controls,javafx.fxml ^
   -cp ".;..\libs\sqlite-jdbc-3.47.1.0.jar" DisplayQueryResults
```

Executing a New Query

When you run the app, it displays a default `SELECT` query. It executes that query to retrieve the `authors` table's data then displays the data in the `TableView`. You can enter your own query in the `queryTextArea` then press or select the Submit Query `Button` to execute it and display its results. The following screen capture contains the results of a query that combines authors with the titles

and edition numbers of the books they wrote:

Sorting a TableView's Data

You can sort a `TableView`'s data by a specific column in ascending or descending order. When you click a column's name, the `TableView` sorts the rows in ascending order based on that column's data. If you click the same column's name again, it sorts the rows in descending order by that column's data. The following screen capture contains the same results after clicking the edition column's title twice to sort the rows in descending order by edition number:

20.6.2 `DisplayQueryResultsController` Class

Class `DisplayQueryResultsController` (Fig. 20.2) responds to the GUI's events and updates the GUI accordingly. We do not present the JavaFX `Application` subclass here (located in `DisplayQueryResults.java`) because it performs the same tasks you've used previously to load the app's FXML GUI and initialize the controller. We split the program into parts for discussion purposes. Lines 3–19 import the classes and interfaces we use in this example.

Click here to view code image

```
1  // Fig. 20.2: DisplayQueryResultsController.java
2  // Controller for the DisplayQueryResults app
3  import java.sql.Connection;
4  import java.sql.DriverManager;
5  import java.sql.ResultSet;
6  import java.sql.ResultSetMetaData;
7  import java.sql.SQLException;
8  import java.sql.Statement;
9  import java.nio.file.Path;
10 import javafx.beans.property.SimpleObjectProperty;
11 import javafx.collections.FXCollections;
12 import javafx.collections.ObservableList;
13 import javafx.event.ActionEvent;
14 import javafx.fxml.FXML;
15 import javafx.scene.control.Alert;
16 import javafx.scene.control.Alert.AlertType;
17 import javafx.scene.control.TableColumn;
18 import javafx.scene.control.TableView;
19 import javafx.scene.control.TextArea;
20
```

Fig. 20.2 | Controller for the `DisplayQueryResults` app.

Instance Variables and `static` Variables

Lines 22–23 define the instance variables for programmatically interacting with the GUI's `TextArea` and `TableView`. Lines 26–28 define the JDBC URL for accessing the `books.db` database. Line 31 defines the default query to display and execute when you run the app. Line 33 defines the `Connection` variable used to manage the database connection.

Click here to view code image

```
21  public class DisplayQueryResultsController {
22     @FXML private TextArea queryTextArea;
23     @FXML private TableView<ObservableList<Object>> tableView;
24
25     // books.db location
26     private static final String DATABASE_URL = "jdbc:sqlite:" +
27        Path.of(System.getProperty("user.home"),
28           "Documents", "examples", "ch20", "books.db");
29
30     // default query retrieves all data from authors table
31     private static final String DEFAULT_QUERY = "SELECT * FROM authors";
32
33     private Connection connection;
34
```

Method `initialize`

When the `FXMLLoader` calls the controller's `initialize` method, it attempts to execute the default query. If that fails, lines 44–48 catch the exception, display an `Alert` dialog (by calling method `displayAlert` in lines 117–123), close the connection and terminate the app. Closing the connection also closes any associated `Statement`s and `ResultSet`s.

Click here to view code image

```
35     // display DEFAULT_QUERY, connect to books.db and execute DEFAULT_QUERY
36     @FXML
37     public void initialize() {
38        queryTextArea.setText(DEFAULT_QUERY);
39
40        try {
41           connection = DriverManager.getConnection(DATABASE_URL);
42           executeQuery(DEFAULT_QUERY);
```

```
43          }
44          catch (SQLException sqlException) {
45             displayAlert(AlertType.ERROR, "Database Error",
46                sqlException.getMessage());
47             System.exit(1); // terminate application
48          }
49       }
50
```

Method `submitQueryButtonPressed`

When the user clicks the Submit Query `Button`, method `submitQueryButtonPressed` (lines 52–55) invokes method `executeQuery`, passing the text in `queryTextArea`.

Click here to view code image

```
51       // query the database and display results in TableView
52       @FXML
53       void submitQueryButtonPressed(ActionEvent event) {
54          executeQuery(queryTextArea.getText());
55       }
56
```

Method `executeQuery`

The `executeQuery` method (lines 59–114) queries the database, processes the results and dynamically updates the `TableView` to display the data. Line 60 uses the `connection` object to create a `Statement`. Line 61 uses the `Statement` to execute the SQL query and obtain a `ResultSet` containing the retrieved data. The `Statement` and `ResultSet` will be closed automatically when the program exits the `try` block. If a `SQLException` occurs during this process, it's caught and displayed to the user using an `Alert` dialog (lines 110–113).

Click here to view code image

```
57       // performs the query and updates the ObservableList<Object>
58       // that the TableView uses to populate its rows and columns
59       private void executeQuery(String query) {
60          try (Statement statement = connection.createStatement();
61               ResultSet resultSet = statement.executeQuery(query)) {
62             tableView.getColumns().clear(); // remove current content
63
64             // create two-dimensional ObservableList for current ResultSet
65             ObservableList<ObservableList<Object>> data =
```

```
66              FXCollections.observableArrayList();
67
68          // use the ResultSetMetaData to get the column heads
69          // and set up the TableView's columns
70          ResultSetMetaData metaData = resultSet.getMetaData();
71          int columnCount = metaData.getColumnCount();
72
73          for (int i = 1; i <= columnCount; i++) {
74              // create a new TableColumn for each column in the ResultSet
75              var column = new TableColumn<ObservableList<Object>, Object>(
76                  metaData.getColumnName(i));
77
78              // map the column to the corresponding index in each row
79              final int columnIndex = i - 1;
80              column.setCellValueFactory(param ->
81                  new SimpleObjectProperty<>(
82                      param.getValue().get(columnIndex)));
83
84              // autoscale column width to divide space equally
85              column.prefWidthProperty().bind(
86                  tableView.widthProperty().divide(columnCount));
87
88              // add the column to the TableView
89              tableView.getColumns().add(column);
90          }
91
92          // populate the rows from the ResultSet
93          while (resultSet.next()) {
94              // create a new ObservableList to represent the row
95              ObservableList<Object> row =
96                  FXCollections.observableArrayList();
97
98              // populate the row with data from the ResultSet
99              for (int i = 1; i <= columnCount; i++) {
100                 row.add(resultSet.getObject(i));
101             }
102
103             // add the row to the data list
104             data.add(row);
105         }
106
107         // set the TableView's items to the ObservableList data
108         tableView.setItems(data);
109     }
110     catch (SQLException sqlException) {
111         displayAlert(AlertType.ERROR, "Query Error",
112             sqlException.getMessage());
113     }
114 }
115
```

Clearing the `TableView` and Getting the `ResultSetMetaData`

If the query is successful, the method processes the `ResultSet`. Line 62 removes any existing columns in the `TableView` to prepare to display the new query results. Lines 65–66 create the empty `ObservableList data`, which will hold each row of the query results as an `ObservableList<Object>`. Line 70 gets the `ResultSetMetaData` so we can get the number of columns (line 71) and the column names (line 76).

Setting Up the `TableView`'s `TableColumn`s

Lines 73–90 set up the `TableView`'s **`TableColumn`s**, which display the `ResultSet`'s data. Lines 75–76 create a new `TableColumn` object for each `ResultSet` column and set the `TableColumn`'s header to the corresponding `ResultSet` column's name retrieved from the `ResultSetMetaData`. `TableColumn` is a generic class:

- The first type argument—`ObservableList<Object>`—tells the `TableColumn` the data structure type that represents a row in the `TableView`.
- The second type argument—`Object`—tells the `TableColumn` the data type each cell in the column displays.

For simplicity, we treat all the `ResultSet` data as type `Object`. The `TableColumn`s will fill the cells with the `Object`s' `String` representations.

Lines 79–82 bind each `TableColumn`'s cells to the `ObservableList`s representing the rows:

- The `columnIndex` (line 79) determines the `ObservableList` index for a given `TableColumn`'s data—recall that `ResultSet`s are indexed from `1`, but `ObservableList`s are indexed from `0`.
- A `TableView` relies on observable properties to determine when to update its cells and what data to display. **`TableColumn` method `setCellValueFactory`** receives a lambda that describes how to retrieve a value for this column from its rows' `ObservableList`s of data. The lambda's parameter receives a `TableView` row and uses that object's `getValue` method to get the `ObservableList<Object>` bound to that row—we'll populate those from the `ResultSet`

momentarily. We call the `ObservableList`'s `get` method to get the data at location `columnIndex`, then wrap this value in a `SimpleObjectProperty`—an observable property that stores an `Object`.

Lines 85–86 bind the column widths to the `TableView`'s width divided by the number of columns to scale the column widths to fit in the `TableView`. Line 89 adds the `TableColumn` to the `TableView`.

Populating the `ObservableList` `data` with the `ResultSet`

Lines 93–105 use the `ResultSet`'s data to populate the `ObservableList` `data`. Lines 95– 96 create an `ObservableList<Object>` to store the current row's data. Lines 99–101 iterate through the `ResultSet` columns, adding each column's value to `row`. Then, line 104 adds that row to the `ObservableList` `data`. After processing all the `ResultSet` rows, line 108 uses the **`TableView`'s `setItems` method** to bind the `TableView` to `data`, which automatically updates the displayed data.

Method `displayAlert`

When an exception occurs, the app calls method `displayAlert` (lines 117–123) to create and display an **`Alert` dialog** (package `javafx.scene.control`) containing a message. Line 119 creates the dialog, passing its `AlertType` to the constructor. The `AlertType.ERROR` constant displays a dialog with a red icon containing an X to indicate an error. Line 120 sets the title that appears in the dialog's title bar. Line 121 sets the message that appears inside the dialog. Finally, line 122 calls `Alert` method `showAndWait`, which makes this a modal dialog. The user must close the dialog before interacting with the rest of the app.

Click here to view code image

```
116     // display an Alert dialog
117     private void displayAlert(
118        AlertType type, String title, String message) {
119        Alert alert = new Alert(type);
120        alert.setTitle(title);
121        alert.setContentText(message);
122        alert.showAndWait();
123     }
124  }
```

20.7 PreparedStatements

A **PreparedStatement** enables you to create compiled SQL statements that execute more efficiently than Statements, especially executing SQL statements multiple times. Prepared-Statements also can specify parameters, making them more flexible than Statements. You can execute the same query repeatedly with different parameter values. For example, in the books database, you might want to locate all book titles for an author with a specific last and first name, and you might want to execute that query for several authors. With a Prepared-Statement, that query is defined as:

Click here to view code image

```
PreparedStatement authorBooks = connection.prepareStatement("""
   SELECT last, first, title FROM authors
   INNER JOIN author_ISBN ON authors.id=author_ISBN.id
   INNER JOIN titles ON titles.isbn=author_ISBN.isbn
   WHERE last = ? AND first = ?""");
```

The question marks (?) in the SQL statement's last line are placeholders for values passed to the database as part of the query. Before executing a PreparedStatement, the program must specify the values by using the PreparedStatement interface's *set* methods.

Setting Parameter Values

For the preceding query, both parameters are Strings that can be set with **Prepared-Statement method setString** as follows:

Click here to view code image

```
authorBooks.setString(1, "Deitel");
authorBooks.setString(2, "Paul");
```

Method setString's first argument represents the parameter number being set, and the second argument is that parameter's value. **Parameter numbers are counted from 1, starting with the first question mark (?).** When the program executes the preceding Prepared-Statement with the parameter values set above, the SQL passed to the database is

Click here to view code image

```
SELECT last, first, title FROM authors
INNER JOIN author_ISBN ON authors.id=author_ISBN.id
INNER JOIN Titles ON titles.isbn=author_ISBN.isbn
WHERE last = 'Deitel' AND first = 'Paul'
```

Method `setString` automatically escapes `String` parameter values as necessary. For example, if the last name is O'Brien, the statement

Click here to view code image

```
authorBooks.setString(1, "O'Brien");
```

escapes the `'` character in O'Brien by replacing it with two single-quote characters so the `'` appears correctly in the database.

`PreparedStatement`s Are More Secure

`PreparedStatement`s help prevent SQL injection attacks,[4] which typically occur in SQL statements that include improper user input. To avoid this security issue, use `PreparedStatement`s in which user input can be supplied only via parameters—each indicated with a question mark (`?`) in a `PreparedStatement`. Once you've created such a `PreparedStatement`, you can use its *set* methods to specify the user input as arguments for those parameters.

4. "SQL Injection." Accessed January 13, 2025. https://owasp.org/www-community/attacks/SQL_Injection.

Interface `PreparedStatement` provides *set* methods for each supported SQL type. Use the *set* method that matches the parameter's SQL type in the database—`SQLException`s occur when a program attempts to convert a parameter value to an incorrect type.

20.7.1 `AddressBook` App That Uses `PreparedStatement`s

We now present an `AddressBook` JavaFX app that enables you to browse existing entries, add new entries and search for entries with a last name that begins with the specified characters. Our `addressbook` SQLite database contains an `addresses` table with the columns `id`, `first`, `last`, `email` and `phone`. The column `id` is an auto-incremented identity column in the `addresses` table.

Creating the `addressbook` Database

In your command line, change to the `fig20_03-05` folder. Create the `addressbook.db` database using the `sqlite3` command as follows:

- On macOS or in a Windows Command Prompt, enter the following command:

  Click here to view code image

  ```
  sqlite3 addressbook.db < addressbook.sql
  ```

- In a Windows PowerShell, enter the following command to read `books.sql` then provide it as input to the `sqlite3` command:

  Click here to view code image

  ```
  Get-Content addressbook.sql | sqlite3.exe addressbook.db
  ```

20.7.2 `record` Class `Person`

Our `AddressBook` app loads data into `Person` objects (Fig. 20.3). Each represents one entry in the `addressbook` database. The `record` class contains instance variables for the address ID, first name, last name, email address and phone number, as well as a custom `toString` method that returns the `Person`'s name in the format

```
last name, first name
```

We do not use the address ID in this example.

Click here to view code image

```
1  // Fig. 20.3: Person.java
2  // Person record class that represents an entry in an address book.
3  public record Person (
4     int id, String first, String last, String email, String phone) {
5
6     // custom String representation
7     @Override
8     public String toString() {
9        return String.format("%s, %s", last(), first());
10    }
11 }
```

Fig. 20.3 | `Person record` class that represents an entry in an address book.

20.7.3 Class `PersonQueries`

Class `PersonQueries` (Fig. 20.4) manages the Address Book program's database connection and creates the `PreparedStatement`s for interacting with the database. We split the code into parts for discussion purposes. Lines 19–21 declare three `PreparedStatement` variables.

Click here to view code image

```
1  // Fig. 20.4: PersonQueries.java
2  // PreparedStatements used by the AddressBook program.
3  import java.nio.file.Path;
4  import java.sql.Connection;
5  import java.sql.DriverManager;
6  import java.sql.PreparedStatement;
7  import java.sql.ResultSet;
8  import java.sql.SQLException;
9  import java.util.List;
10 import java.util.ArrayList;
11
12 public class PersonQueries {
13    // addressbook.db location
14    private static final String DATABASE_URL = "jdbc:sqlite:" +
15       Path.of(System.getProperty("user.home"), "Documents",
16          "examples", "ch20", "fig20_03-05", "addressbook.db");
17
18    private Connection connection; // manages connection
19    private PreparedStatement selectAllPeople;
20    private PreparedStatement selectPeopleByLastName;
21    private PreparedStatement insertNewPerson;
22
```

Fig. 20.4 | `PreparedStatements` used by the Address Book program.

Creating `PreparedStatement`s

The constructor (lines 24–47) connects to the database at line 26. Lines 29–30 invoke **`Connection` method `prepareStatement`** to create the `PreparedStatement selectAll-People` that selects all the rows in the `addresses` table and sorts them by last name, then by first name. Lines 34–36 create the `PreparedStatement selectPeopleByLastName` with a parameter. This statement uses the SQL `LIKE` operator to search the `addresses` table by last name. The `?` character specifies the last-name parameter—the text we'll set as this parameter's value will end with a `%` symbol to locate entries for

last names starting with the characters the user enters. Lines 39–42 create the `PreparedStatement` `insertNewPerson` with four parameters that represent the first name, last name, email address and phone number for a new entry. Again, notice the `?` characters used to represent these parameters.

Click here to view code image

```
23      // constructor
24      public PersonQueries() {
25         try {
26            connection = DriverManager.getConnection(DATABASE_URL);
27
28            // create query that selects all entries in the AddressBook
29            selectAllPeople = connection.prepareStatement(
30               "SELECT * FROM addresses ORDER BY last, first");
31
32            // create query that selects entries with last names
33            // that begin with the specified characters
34            selectPeopleByLastName = connection.prepareStatement("""
35               SELECT * FROM addresses WHERE last LIKE ?
36               ORDER BY last, first""");
37
38            // create insert operation that adds a new entry to the database
39            insertNewPerson = connection.prepareStatement("""
40               INSERT INTO addresses (first, last, email, phone)
41               VALUES (?, ?, ?, ?)""");
42         }
43         catch (SQLException sqlException) {
44            sqlException.printStackTrace();
45            System.exit(1);
46         }
47      }
48
```

`PersonQueries` Method `getAllPeople`

Method `getAllPeople` (lines 50–71) executes `PreparedStatement` `selectAllPeople` (line 52) by calling **`PreparedStatement` method `executeQuery`**, which returns a `ResultSet` containing the rows that match the query—in this case, all the rows in the `addresses` table. Lines 53–62 place the query results in an `ArrayList<Person>`, which is returned to the caller at line 64.

Click here to view code image

```
49      // select all of the addresses in the database
50      public List<Person> getAllPeople() {
```

```
51          // executeQuery returns ResultSet containing matching entries
52          try (ResultSet resultSet = selectAllPeople.executeQuery()) {
53             List<Person> results = new ArrayList<>();
54
55             while (resultSet.next()) {
56                results.add(new Person(
57                   resultSet.getInt("id"),
58                   resultSet.getString("first"),
59                   resultSet.getString("last"),
60                   resultSet.getString("email"),
61                   resultSet.getString("phone")));
62             }
63
64             return results;
65          }
66          catch (SQLException sqlException) {
67             sqlException.printStackTrace();
68          }
69
70          return null;
71       }
72
```

`PersonQueries` Method `getPeopleByLastName`

Method `getPeopleByLastName` (lines 74–102) uses `PreparedStatement` method `set-String` to set the parameter of `selectPeopleByLastName` (line 76). Then, line 84 executes the query and lines 85–94 place the query results in an `ArrayList<Person>`. Line 96 returns the `ArrayList` to the caller.

Click here to view code image

```
73       // select person by last name
74       public List<Person> getPeopleByLastName(String lastName) {
75          try {
76             selectPeopleByLastName.setString(1, lastName); // set last name
77          }
78          catch (SQLException sqlException) {
79             sqlException.printStackTrace();
80             return null;
81          }
82
83          // executeQuery returns ResultSet containing matching entries
84          try (ResultSet resultSet = selectPeopleByLastName.executeQuery()) {
85             List<Person> results = new ArrayList<>();
86
87             while (resultSet.next()) {
88                results.add(new Person(
```

```
89                  resultSet.getInt("id"),
90                  resultSet.getString("first"),
91                  resultSet.getString("last"),
92                  resultSet.getString("email"),
93                  resultSet.getString("phone")));
94           }
95
96           return results;
97        }
98        catch (SQLException sqlException) {
99           sqlException.printStackTrace();
100          return null;
101       }
102    }
103
```

`PersonQueries` Methods `addPerson` and `close`

Method `addPerson` (lines 105–122) uses `PreparedStatement` method `setString` (lines 111–114) to set the `insertNewPerson` `PreparedStatement`'s parameters. Line 116 uses **`PreparedStatement` method `executeUpdate`** to update the database by inserting the new record. This method returns an `int` indicating the number of rows that were updated (or inserted) in the database. Method `close` (lines 125–132) closes the database connection.

Click here to view code image

```
104    // add an entry
105    public int addPerson(String first, String last,
106       String email, String phone) {
107
108       // insert the new entry; returns # of rows updated
109       try {
110          // set parameters
111          insertNewPerson.setString(1, first);
112          insertNewPerson.setString(2, last);
113          insertNewPerson.setString(3, email);
114          insertNewPerson.setString(4, phone);
115
116          return insertNewPerson.executeUpdate();
117       }
118       catch (SQLException sqlException) {
119          sqlException.printStackTrace();
120          return 0;
121       }
122    }
123
124    // close the database connection
```

```
125     public void close() {
126        try {
127           connection.close();
128        }
129        catch (SQLException sqlException) {
130           sqlException.printStackTrace();
131        }
132     }
133  }
```

20.7.4 `AddressBook` GUI

For your convenience, the file `addressbookcommands.txt` (located in the `fig20_03-05` example folder) contains the `javac` and `java` commands for macOS and Windows. The following diagram contains the app's initial GUI (defined in `AddressBook.fxml`) labeled with its fx:ids and populated with `addressbook.db` two default contacts that we included in the `addressbook.sql` SQL script:
Here, we point out only the key elements and their event-handler methods, which we'll present in class `AddressBookController` (Fig. 20.5). For the complete layout details, open `AddressBook.fxml` in Scene Builder. The GUI's primary layout is a `BorderPane`. The controller class defines three event-handling methods:

- `addEntryButtonPressed` is called when the Add Entry `Button` is pressed.
- `findButtonPressed` is called when the Find `Button` is pressed.
- `browseAllButtonPressed` is called when the Browse All `Button` is pressed.

The following screen captures demonstrate user interactions with the app:

20.7.5 Class `AddressBookController`

The `AddressBookController` (Fig. 20.5) class uses a `PersonQueries` object to interact with the database. We do not present the JavaFX `Application` subclass here (located in `AddressBook.java`), because it performs the same tasks you've used previously to load the app's FXML GUI and initialize the controller. We split this code into parts for discussion purposes.

Click here to view code image

```
1  // Fig. 20.5: AddressBookController.java
2  // Controller for the AddressBook app
3  import java.util.List;
4
5  import javafx.collections.FXCollections;
6  import javafx.collections.ObservableList;
7  import javafx.event.ActionEvent;
8  import javafx.fxml.FXML;
9  import javafx.scene.control.Alert;
10 import javafx.scene.control.Alert.AlertType;
11 import javafx.scene.control.ListView;
12 import javafx.scene.control.TextField;
13
14 public class AddressBookController {
15    @FXML private ListView<Person> listView; // displays contact names
16    @FXML private TextField firstNameTextField;
17    @FXML private TextField lastNameTextField;
18    @FXML private TextField emailTextField;
19    @FXML private TextField phoneTextField;
20    @FXML private TextField findByLastNameTextField;
21
```

Fig. 20.5 | Controller for the `AddressBook` app.

Instance Variables

Line 23 creates the `PersonQueries` object. We use the same techniques to populate the `ListView` that we used in Section 16.5, so lines 26–27 create an `ObservableList<Person>` named `contactList` to store the `Person` objects returned by the `PersonQueries` object.

Click here to view code image

```
22    // interacts with the database
23    private final PersonQueries personQueries = new PersonQueries();
24
25    // stores list of Person objects that results from a database query
26    private final ObservableList<Person> contactList =
27       FXCollections.observableArrayList();
28
```

Method `initialize`

When the FXMLLoader initializes the controller, method

`initialize` (lines 30–40) performs the following tasks:

- Lines 32–36 register a lambda that implements interface `ChangeListener` to display the selected contact when the user selects a new item in the `ListView`.
- Line 38 binds the `contactList` to the `ListView`—each time `contactList` changes, the `ListView` will update its list of items.
- Line 39 calls method `getAllEntries` (declared in lines 43–46) to get all the entries from the database and place them in the `contactList`.

Click here to view code image

```
29      // populate listView and set up listener for selection events
30      public void initialize() {
31         // when ListView selection changes, display selected person's data
32         listView.getSelectionModel().selectedItemProperty().addListener(
33            (observableValue, oldValue, newValue) -> {
34               displayContact(newValue);
35            }
36         );
37
38         listView.setItems(contactList); // bind to contactsList
39         getAllEntries(); // populates contactList, which updates listView
40      }
41
```

Methods `getAllEntries` and `selectFirstEntry`

When the app executes, when the user clicks the Browse All `Button` and when the user adds a new entry to the database, the app calls `getAllEntries` (lines 43–46). Line 44 calls class `PersonQueries`' `getAllPeople` method to obtain all the database entries. The resulting `List<Person>` is passed to `ObservableList` method `setAll` to replace the `contactList`'s contents. Then, the `ListView` auto-updates its list of items based on `contactList`'s current contents. Next, line 45 selects the first item in the `ListView` by calling method `selectFirstEntry` (lines 49–51). Line 50 selects the `ListView`'s first item to display that contact's data.

Click here to view code image

```
42      // get all the entries from the database to populate contactList
43      private void getAllEntries() {
```

```
44          contactList.setAll(personQueries.getAllPeople());
45          selectFirstEntry();
46       }
47
48       // select first item in listView
49       private void selectFirstEntry() {
50          listView.getSelectionModel().selectFirst();
51       }
52
```

Method `displayContact`

When the user selects an item in the `ListView`, the `ChangeListener` registered in method `initialize` calls `displayContact` (lines 54–67) to display the selected `Person`'s data. If the argument is `null`, the method clears the `TextField`'s contents—we do not use that feature in this example.

Click here to view code image

```
53       // display contact information
54       private void displayContact(Person person) {
55          if (person != null) {
56             firstNameTextField.setText(person.first());
57             lastNameTextField.setText(person.last());
58             emailTextField.setText(person.email());
59             phoneTextField.setText(person.phone());
60          }
61          else {
62             firstNameTextField.clear();
63             lastNameTextField.clear();
64             emailTextField.clear();
65             phoneTextField.clear();
66          }
67       }
68
```

Method `addEntryButtonPressed`

To add a new entry into the database, you can enter the first name, last name, email and phone number (the `id` will autoincrement) in the `TextField`s that display contact information, then press the Add Entry `Button`. Method `addEntryButtonPressed` (lines 70–86) calls `PersonQueries` method `addPerson` (lines 72–74) to add the new entry to the database. Lines 76–83 display an `Alert` dialog indicating whether the new entry was added. Then, line 85 calls `getAllEntries` to obtain the updated database contents and display

the contacts' names in the `ListView`.

Click here to view code image

```
69      // add a new entry
70      @FXML
71      void addEntryButtonPressed(ActionEvent event) {
72         int result = personQueries.addPerson(
73            firstNameTextField.getText(), lastNameTextField.getText(),
74            emailTextField.getText(), phoneTextField.getText());
75
76      if (result == 1) {
77         displayAlert(AlertType.INFORMATION, "Entry Added",
78            "New entry successfully added.");
79      }
80      else {
81         displayAlert(AlertType.ERROR, "Entry Not Added",
82            "Unable to add entry.");
83      }
84
85      getAllEntries();
86   }
87
```

Method `findButtonPressed`

When the user presses the Find `Button`, the app calls method `findButtonPressed` (lines 89–102). Lines 91–92 call `PersonQueries` method `getPeopleByLastName` to search the database. Note that line 92 appends a `%` to the text input by the user. This enables the corresponding SQL query, which contains a `LIKE` operator, to locate last names that begin with the characters the user entered in the `findByLastNameTextField`. If there are several such entries, they're all displayed in the `ListView` when the `contactList` is updated (line 95), and the first one is selected (line 96).

Click here to view code image

```
88      // find entries with the specified last name
89      @FXML
90      void findButtonPressed(ActionEvent event) {
91         List<Person> people = personQueries.getPeopleByLastName(
92            findByLastNameTextField.getText() + "%");
93
94         if (people.size() > 0) { // display all entries
95            contactList.setAll(people);
96            selectFirstEntry();
97         }
```

```
98          else {
99             displayAlert(AlertType.INFORMATION, "Lastname Not Found",
100               "There are no entries with the specified last name.");
101         }
102      }
103
```

Method `browseAllButtonPressed`

When the user presses or selects Browse All, method `browseAllButtonPressed` (lines 105–108) calls `getAllEntries` to get all the database entries and display them in the `ListView`.

Click here to view code image

```
104      // browse all the entries
105      @FXML
106      void browseAllButtonPressed(ActionEvent event) {
107         getAllEntries();
108      }
109
```

Method `displayAlert`

When an exception occurs, the app calls method `displayAlert` (lines 111–117) to create and display an `Alert` dialog (package `javafx.scene.control`) containing a message.

Click here to view code image

```
110      // display an Alert dialog
111      private void displayAlert(
112         AlertType type, String title, String message) {
113         var alert = new Alert(type);
114         alert.setTitle(title);
115         alert.setContentText(message);
116         alert.showAndWait();
117      }
118   }
```

Generative AI

1 Prompt genAIs for examples of SQL injection attacks and how JDBC `Prepared-Statement`s help prevent them.

2 Prompt genAIs for the advantages of using JDBC

20.8 Stored Procedures

Many database-management systems can store individual or sets of SQL statements in a database, so programs can invoke them. JDBC enables programs to invoke such **stored procedures** using objects that implement the **`CallableStatement` interface**, which can receive arguments specified by the methods inherited from interface `PreparedStatement`. In addition, `CallableStatement`s can specify **output parameters** in which a stored procedure can place return values. The `CallableStatement` interface includes methods to specify which parameters in a stored procedure are output parameters. The interface also includes methods to obtain the values of output parameters returned from a stored procedure.

Although the syntax for creating stored procedures differs across database management systems, the interface `CallableStatement` provides a uniform interface for specifying input and output parameters for stored procedures and for invoking stored procedures.

According to the Java API documentation for interface `CallableStatement`, for maximum portability between database systems, programs should process the update counts (which indicate how many rows were updated) or `ResultSet`s returned from a `Callable-Statement` before obtaining the values of any output parameters.

Generative AI

1 Prompt genAIs for the pros and cons of using stored procedures in Java JDBC applications.

20.9 Transaction Processing

Many database applications require guarantees that a series of database insertions, updates and deletions are executed properly before the application continues processing the next database operation. For example, when you transfer money electronically between bank accounts, several factors determine if the transaction is successful. You begin by specifying the source account and the amount you wish to transfer from that account to a destination

account. Next, you specify the destination account. The bank checks the source account to determine whether its funds are sufficient to complete the transfer. If so, the bank withdraws the specified amount and, if all goes well, deposits it into the destination account to complete the transfer. What happens if the transfer fails after the bank withdraws the money from the source account? In a proper banking system, the bank redeposits the money in the source account. How would you feel if the money was subtracted from your source account and the bank did not deposit the money in the destination account?

Transaction processing enables a program that interacts with a database to treat a database operation (or set of operations) as a single operation. Such an operation also is known as an **atomic operation** or a **transaction**. At the end of a transaction, a decision can be made either to **commit the transaction** or **roll back the transaction**. Committing the transaction finalizes the database operation(s); all insertions, updates and deletions performed as part of the transaction cannot be reversed without performing a new database operation. Rolling back the transaction leaves the database in its state prior to the database operation. This is useful when a portion of a transaction fails to complete properly. In our bank-account-transfer discussion, the transaction would be rolled back if the deposit could not be made into the destination account.

Java provides transaction processing via methods of interface `Connection`. **Method `setAutoCommit`** specifies whether each SQL statement commits after it completes (a `true` argument) or whether several SQL statements should be grouped as a transaction (a `false` argument). If the argument to `setAutoCommit` is `false`, the program must follow the last SQL statement in the transaction with a call to **`Connection` method `commit`** (to commit the changes to the database) or **`Connection` method `rollback`** (to return the database to its state prior to the transaction). Interface `Connection` also provides **method `getAutoCommit`** to determine the auto-commit state for the `Connection`.

Generative AI

1 Prompt genAIs to provide examples of scenarios in which transaction processing is used in real-world applications.

2 Prompt genAIs with the code for Section 20.7's AddressBook app, asking them to add JDBC transaction processing.

20.10 Wrap-Up

This chapter introduced accessing and manipulating relational databases using JDBC. We presented the core concepts of relational databases, including their table, row and column structure, and the importance of primary and foreign keys in maintaining data integrity.

You created and configured SQLlite databases using predefined SQL scripts. We discussed the standard SQL language for querying and manipulating relational databases. We demonstrated the SQL commands `SELECT`, `INSERT`, `UPDATE` and `DELETE` and the clauses `WHERE`, `ORDER BY` and `INNER JOIN` using the SQLite command-line tools in the context of a books database.

Next, we used JDBC to interact with SQLite databases from Java applications. We used class `DriverManager` to obtain a database `Connection`, and used it to create `Statement`s containing queries that interacted with the database's data. We then executed the `Statement`s and processed their `ResultSet`s. For each `ResultSet`, we used the `ResultSetMetaData` to get and display the column names in the query results.

The AddressBook app demonstrated CRUD (Create, Read, Update, Delete) operations. We used `PreparedStatement`s to create precompiled SQL statements that accept parameters, improving performance. We incorporated a JavaFX `TableView` into a GUI and bound a `ResultSet` to the `TableView` via JavaFX observable collections. We also discussed how `PreparedStatement`s enhance security by preventing SQL injection attacks.

We discussed stored procedures and `CallableStatement`s for encapsulating SQL logic within the database for reuse and efficiency. Finally, we discussed transaction processing, explaining how to group multiple database operations into one atomic transaction. You learned how to use a `Connection` to manage transactions, including committing changes to a database or rolling them back if errors occur.

In the next chapter, you'll learn about the Java Platform Module System (JPMS), which helps developers at all levels be more productive as they build, maintain and evolve software systems, especially large systems.

Special Section: Generative AI Exercises

For each of the following questions, try multiple genAIs to determine which performs best. Always verify the results. Keep in mind that they sometimes make mistakes and even hallucinate, creating unreliable information.

20.1 Prompt genAIs asking which genAIs are best for generating SQL code.

20.2 Prompt genAIs with a description of the tables in Section 20.4's `books` database, asking them to write a SQL script that creates a SQLite database and populates it with sample authors and books. Compare the generated SQL statements to those in this chapter's `books.sql` file.

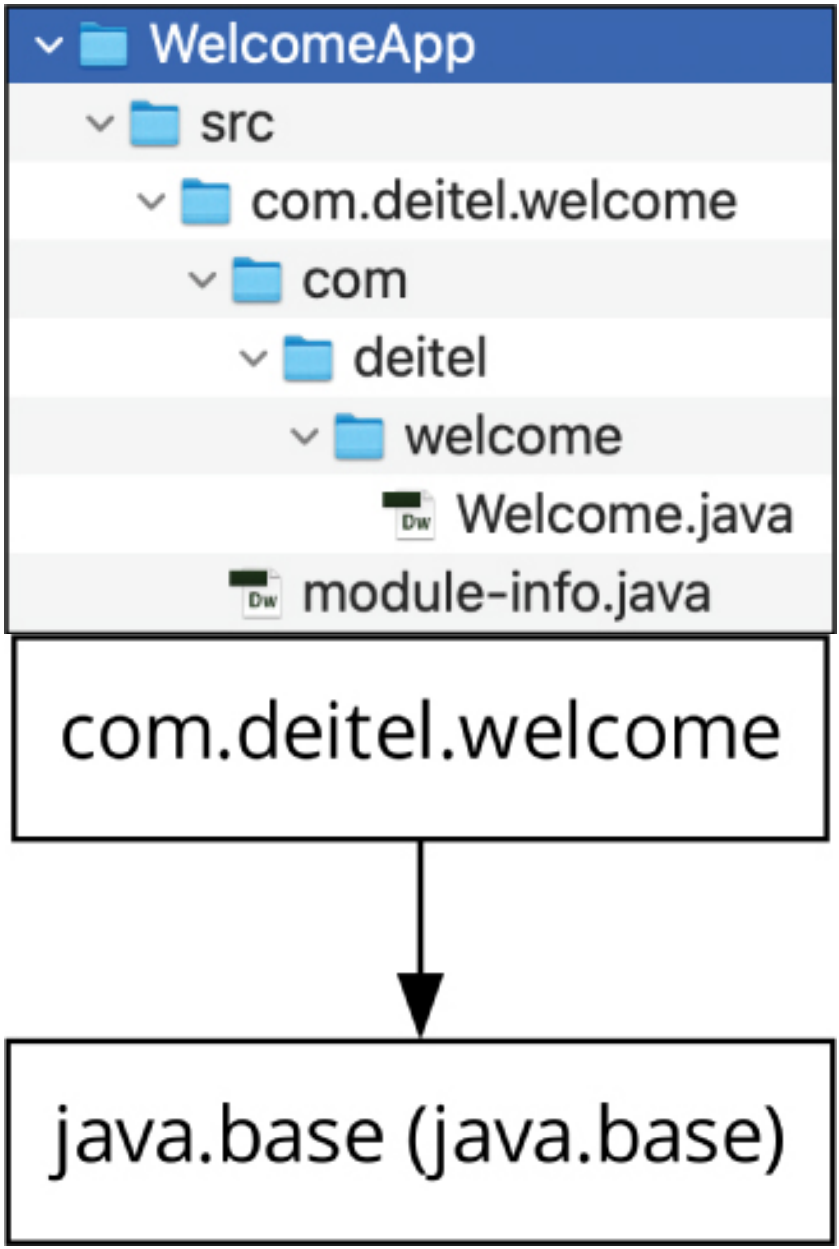
20.3 Prompt genAIs for an overview of the types of database management systems and the kinds of data they're typically used to process.

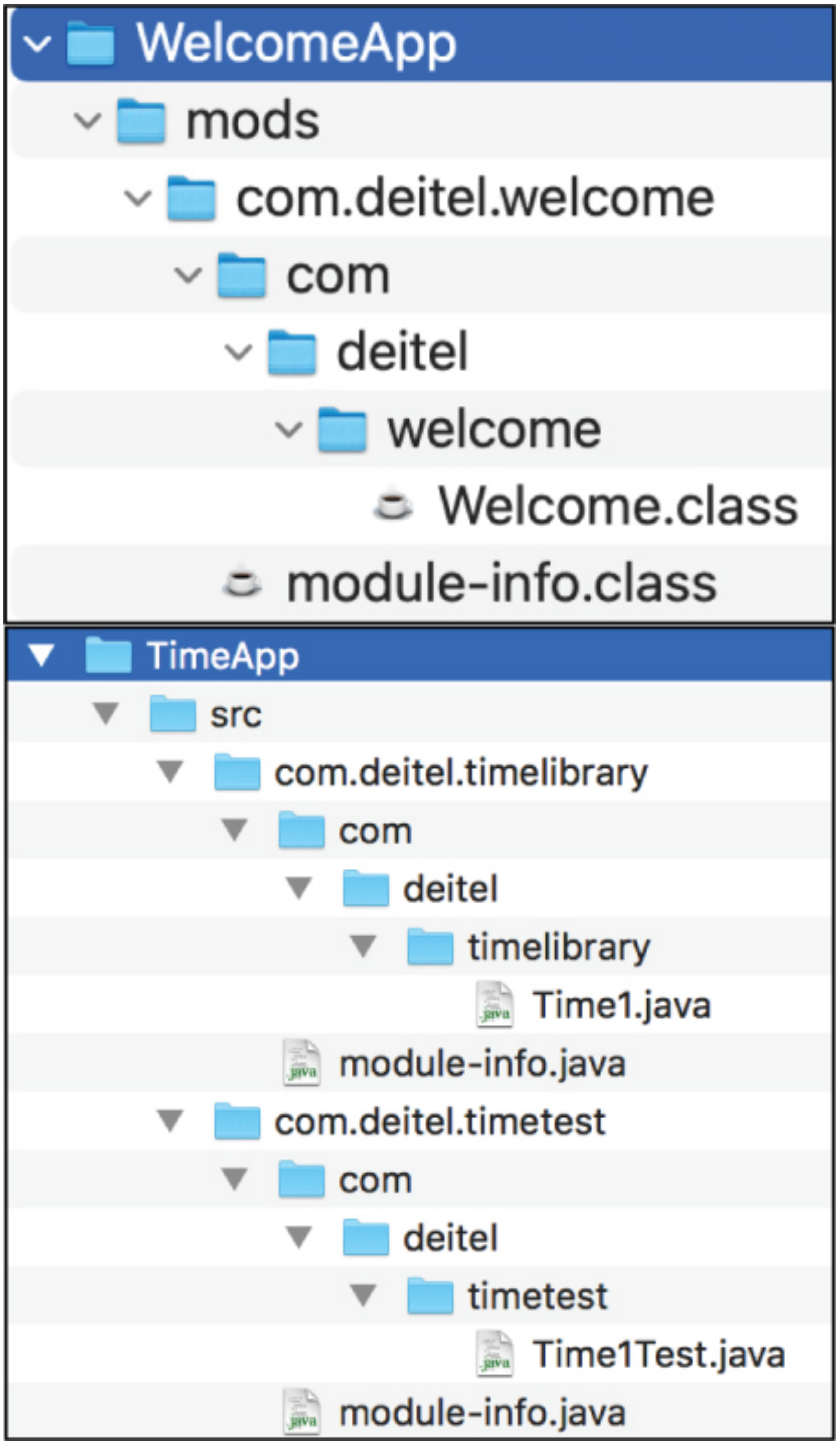
20.4 Prompt genAIs to compare and contrast popular relational database management systems (RDBMSs).

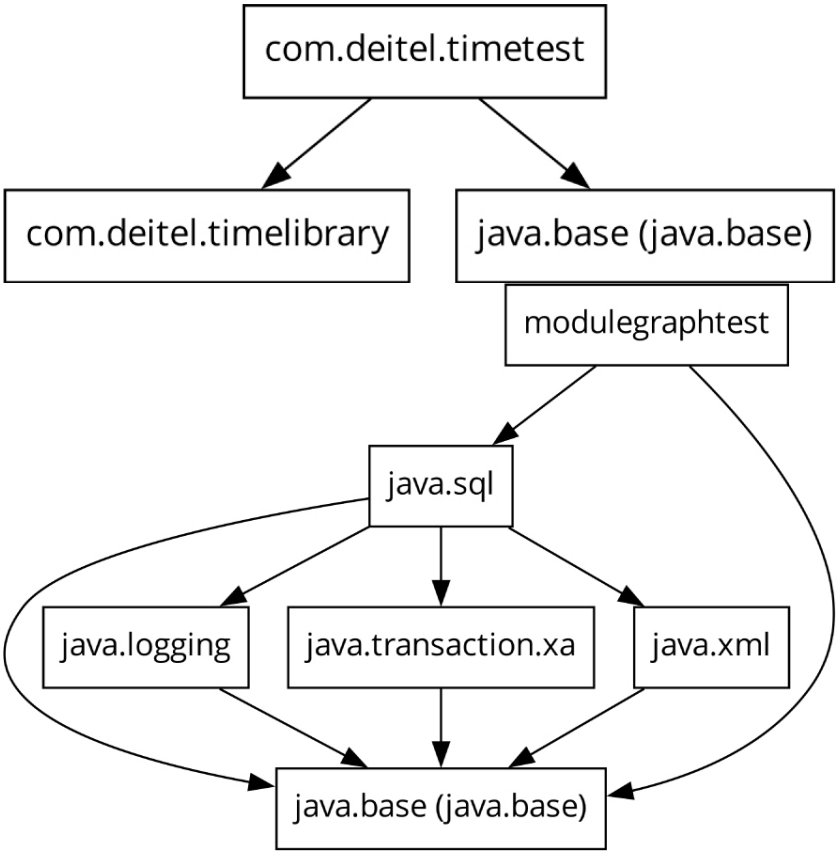
20.5 Databases often contain massive amounts of data. Reading all the data from a table or a joined set of tables can cause a JavaFX GUI to become unresponsive. Prompt genAIs to explain how to implement pagination for large JDBC `ResultSet`s in a JavaFX `TableView`.

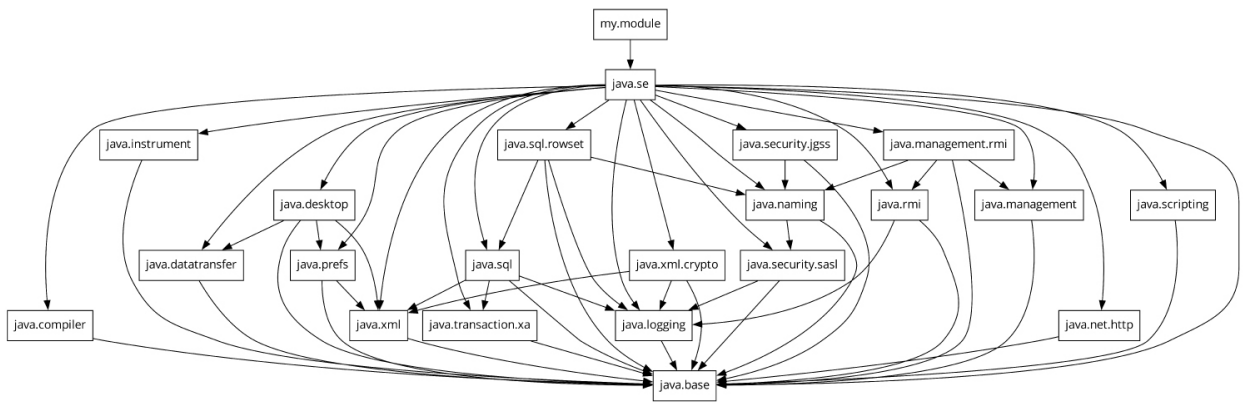
20.6 Prompt genAIs for a list of best practices to ensure JavaFX applications remain responsive when interacting with and displaying data from large databases.

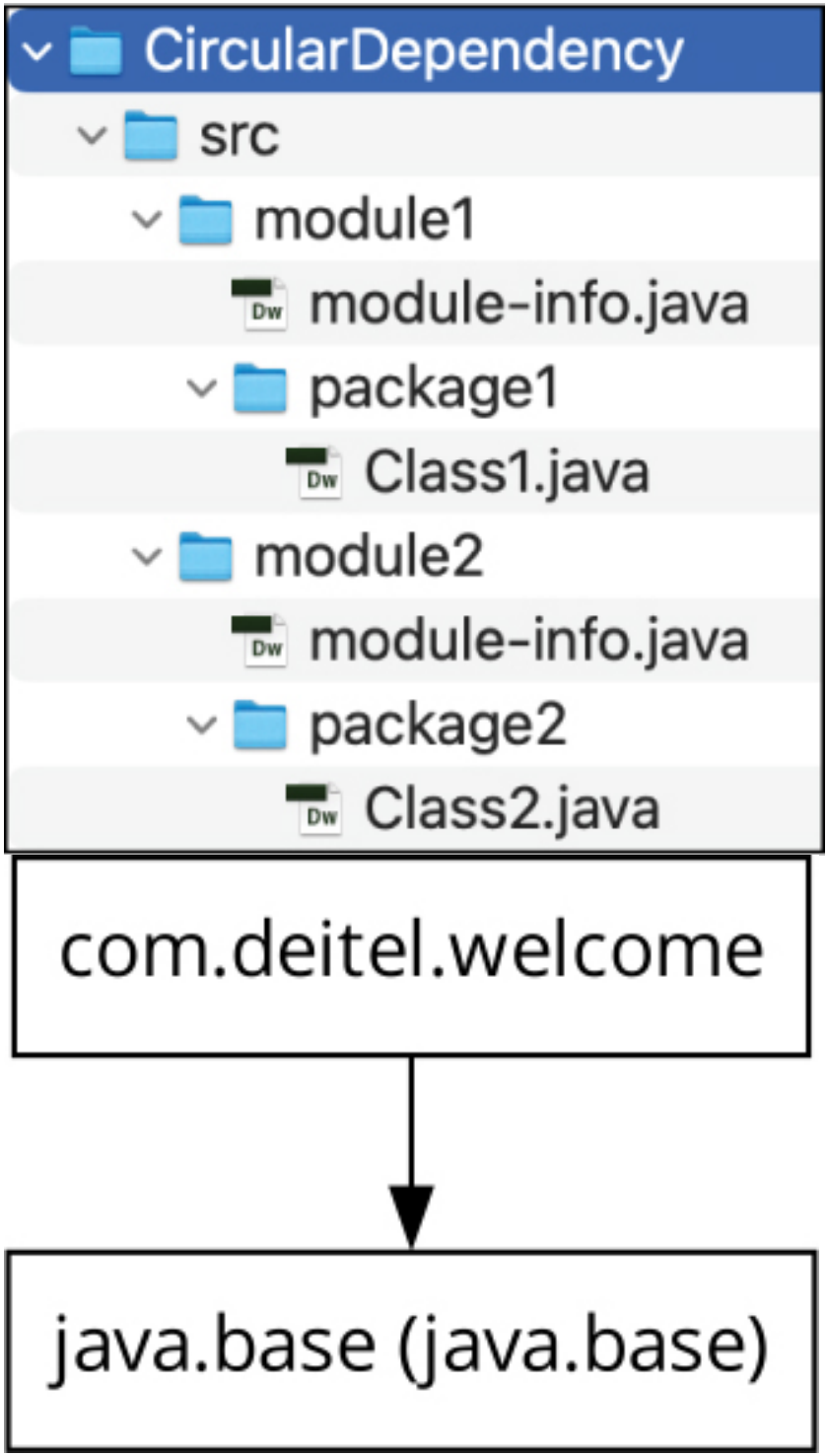
20.7 There are sometimes differences in the SQL used to create databases for various RDBMS. If you prefer to work with an RDBMS other than SQLite, prompt genAIs with our SQLite SQL scripts and ask them to update the scripts for use with your preferred RDBMS. Also, ask for step-by-step instructions on running the scripts to create the databases.

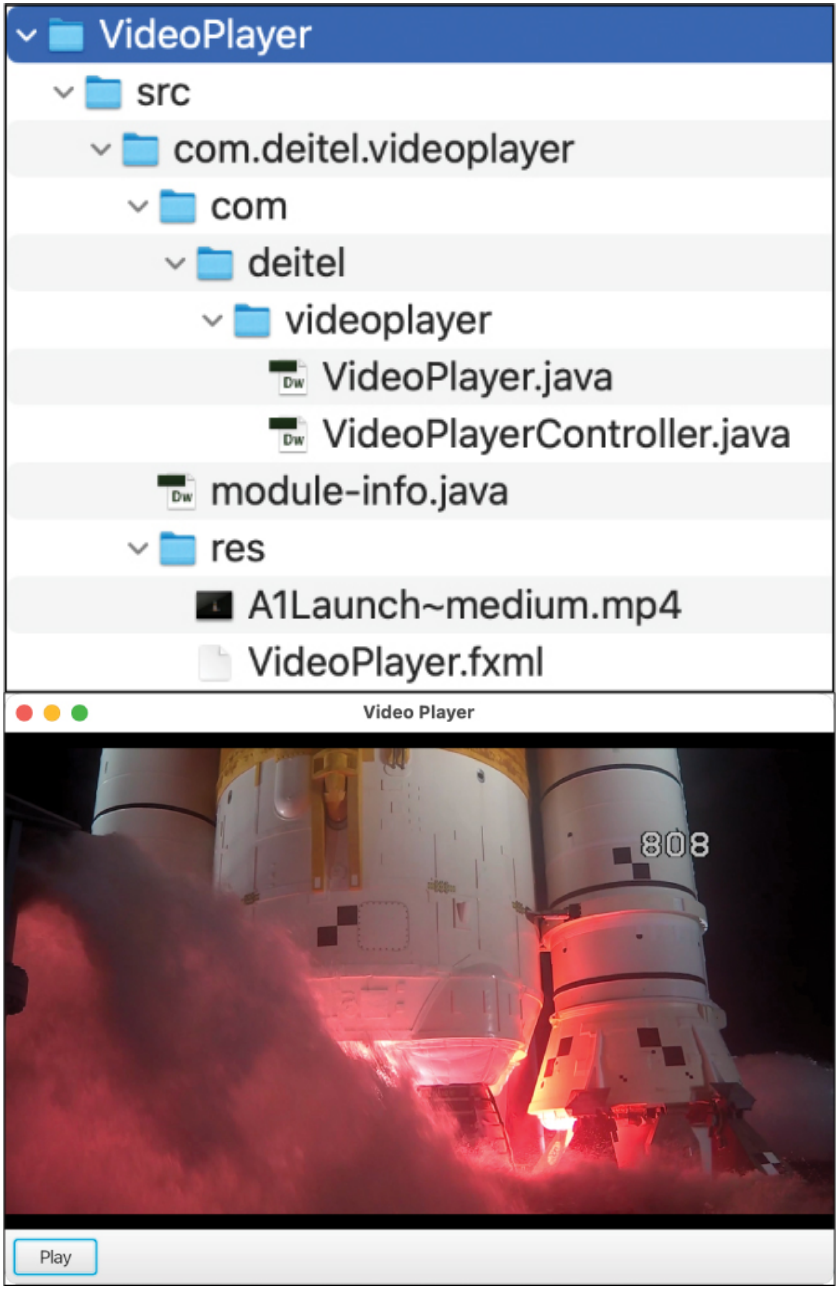

WelcomeApp
src
com.deitel.welcome
com
deitel
welcome
Welcome.java
module-info.java
com.deitel.welcome
java.base (java.base)

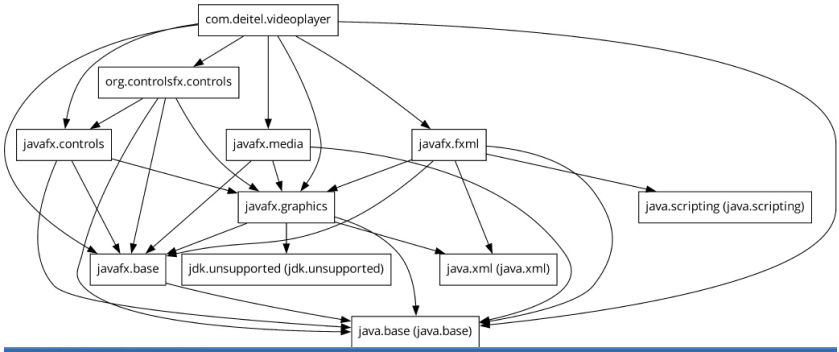

WelcomeApp
mods
com.deitel.welcome
com
deitel
welcome
Welcome.class
module-info.class
TimeApp
src
com.deitel.timelibrary
com
deitel
timelibrary
Time1.java
module-info.java
com.deitel.timetest
com
deitel
timetest
Time1Test.java
module-info.java

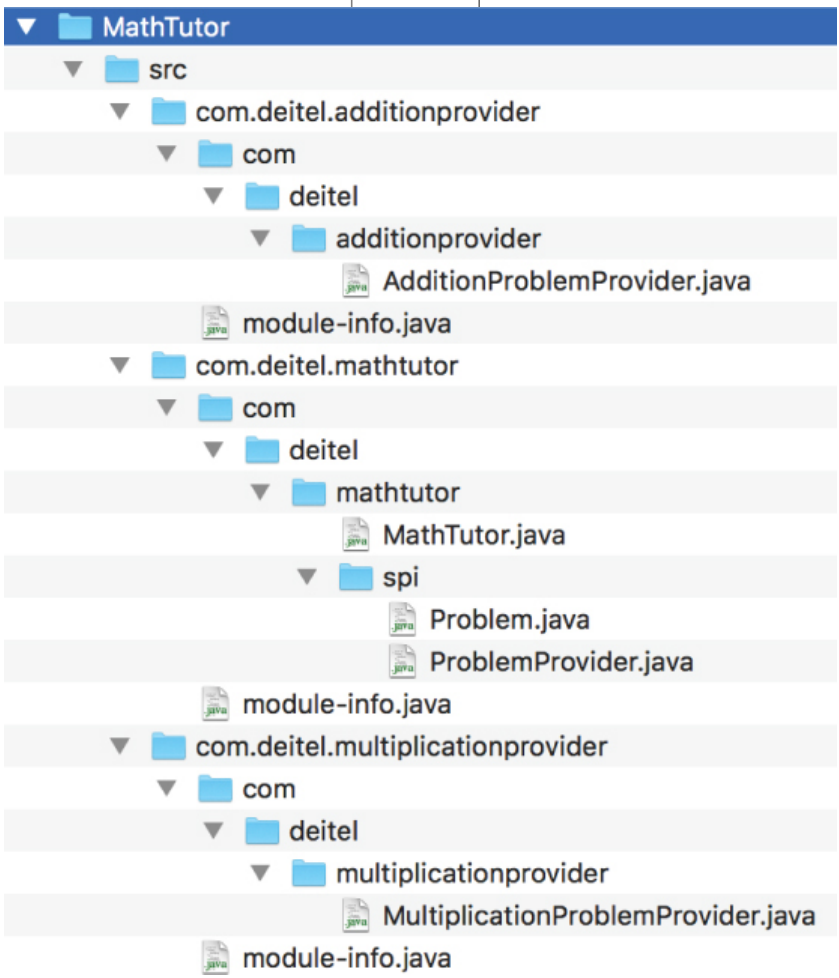

com.deitel.timetest
com.deitel.timelibrary
java.base (java.base)
modulegraphtest
java.sql
java.logging
java.transaction.xa
java.xml
java.base (java.base)


my.module
java.se
java.instrument
java.sql.rowset
java.security.jgss
java.management.rmi
java.desktop
java.naming
java.rmi
java.management
java.scripting
java.datatransfer
java.prefs
java.sql
java.xml.crypto
java.security.sasl
java.compiler
java.xml
java.transaction.xa
java.logging
java.net.http
java.base


CircularDependency
src
module1
module-info.java
package1
Class1.java
module2
module-info.java
package2
Class2.java
com.deitel.welcome
java.base (java.base)


VideoPlayer
src
com.deitel.videoplayer
com
deitel
videoplayer
VideoPlayer.java
VideoPlayerController.java
module-info.java
res
A1Launch~medium.mp4
VideoPlayer.fxml
Video Player
808
Play


com.deitel.videoplayer
org.controlsfx.controls
javafx.controls
javafx.media
javafx.fxml
javafx.graphics
java.scripting (java.scripting)
javafx.base
jdk.unsupported (jdk.unsupported)
java.xml (java.xml)
java.base (java.base)


MathTutor
src
com.deitel.additionprovider
com
deitel
additionprovider
AdditionProblemProvider.java
module-info.java
com.deitel.mathtutor
com
deitel
mathtutor
MathTutor.java
spi
Problem.java
ProblemProvider.java
module-info.java
com.deitel.multiplicationprovider
com
deitel
multiplicationprovider
MultiplicationProblemProvider.java
module-info.java

Java Platform Module System

Objectives

In this chapter, you'll:

- Understand the motivation for modularity.
- Review the Java Platform Module System JEPs and JSRs.
- Create `module` declarations that specify module dependencies with `requires` and specify which packages a module makes available to other modules with `exports`.
- Allow runtime reflection of types with `open` and `opens`.
- Use services to loosely couple system components to make large-scale systems easier to develop and maintain.
- Indicate that a module uses a service or provides a service implementation with the `uses` and `provides...with` directives, respectively.
- Use the `jdeps` command to determine a module's dependencies.
- Migrate non-modularized code with unnamed and automatic modules.
- Use Graphviz to create module graphs.
- Learn how the runtime determines dependencies with the module resolver.
- Use `jlink` to create smaller runtimes appropriate for resource-constrained devices.

Outline

21.1 Introduction

21.2 Module Declarations

21.2.1 `requires`

21.2.2 `requires transitive`—Implied Readability

21.2.3 `exports` and `exports...to`

21.2.4 `uses`

21.2.5 `provides...with`

21.2.6 `open`, `opens` and `opens... to`,

21.2.7 Restricted Keywords

21.3 Modularized `Welcome` App

21.3.1 `Welcome` App's Structure

21.3.2 Class `Welcome`

21.3.3 `module-info.java`

21.3.4 Module-Dependency Graph

21.3.5 Compiling a Module

21.3.6 Running an App from a Module's Exploded Folders

21.3.7 Packaging a Module into a Modular JAR File

21.3.8 Running the `Welcome` App from a Modular JAR File

21.3.9 Aside: Classpath vs. Module Path

21.4 Creating and Using a Custom Module

21.4.1 Exporting a Package for Use in Other Modules

21.4.2 Using a Class from a Package in Another Module

21.4.3 Compiling and Running the Example

21.4.4 Packaging the App into Modular JAR Files

21.4.5 Strong Encapsulation and Accessibility

21.5 Module-Dependency Graphs: A Deeper Look

21.5.1 `java.sql`

21.5.2 `java.se`

21.5.3 Error: Module Graph with a Cycle

21.6 Migrating Code to Modules
21.6.1 Unnamed Module
21.6.2 Automatic Modules
21.6.3 `jdeps` Tool for Java Dependency Analysis
21.7 Resources in Modules
21.7.1 Requiring Multiple Modules
21.7.2 Opening a Module for Reflection
21.7.3 Compiling the Module
21.7.4 Running a Modularized App
21.7.5 Module-Dependency Graph
21.8 Creating Custom Runtimes with `jlink`
21.8.1 Custom Runtime Containing Only `java.base`
21.8.2 Creating a Custom Runtime for the `Welcome` App
21.8.3 Executing the `Welcome` App Using a Custom Runtime
21.8.4 Using the Module Resolver on a Custom Runtime
21.9 Services and `ServiceLoader`
21.9.1 Service-Provider Interface
21.9.2 Loading and Consuming Service Providers
21.9.3 `uses` Module Directive and Service Consumers
21.9.4 Running the App with No Service Providers
21.9.5 Implementing a Service Provider
21.9.6 `provides...with` Module Directive and Declaring a Service Provider
21.9.7 Running the App with One Service Provider
21.9.8 Implementing a Second Service Provider
21.9.9 Running the App with Two Service Providers
21.10 Wrap-Up
Special Section: Generative AI Exercises

21.1 Introduction[1]

1. We'd like to thank Brian Goetz, Alex Buckley, Alan Bateman, Lance

Anderson, Mandy Chung and Paul Bakker for answering our questions and sharing insights when we wrote this chapter for *Java 9 for Programmers*.

This chapter introduces the **Java Platform Module System (JPMS)**. Modularity—the result of **Project Jigsaw**[2]—helps developers at all levels be more productive as they build, maintain and evolve software systems, especially large systems.

2. "Project Jigsaw." Accessed January 7, 2025. https://openjdk.org/projects/jigsaw/.

What is a Module?

Modularity adds a higher level of aggregation above packages. The key language element is the **module**—a uniquely named, reusable group of related packages, as well as resources (like images and other files) and a **module declaration** specifying:

- the module's name,
- the module's dependencies (that is, other modules this module depends on),
- the packages it explicitly makes available to other modules (all other packages in the module are implicitly unavailable to other modules),
- the services it offers,
- the services it consumes, and
- to what other modules it allows reflection.

History

The Java Platform was first released in 1995. Millions of developers use Java to build everything from small apps for resource-constrained devices—like those in the Internet of Things (IoT) and other embedded devices—to large-scale business-critical and mission-critical systems. Until Java 9, the Java platform was primarily a monolithic, one-size-fits-all solution.

Modularizing the Java Platform was challenging to implement and took many years. *JSR 277: Java Module System* was first proposed in 2005[3] for Java 7. This JSR was later superseded by *JSR*

376: Java Platform Module System and targeted for Java 8. The Java Platform was fully modularized in Java 9 and subsequently refined in Java 16 and 17.

3. "JSR 277: Java Module System." Accessed January 7, 2025. https://jcp.org/en/jsr/detail?id=277.

Goals

According to JSR 376, the key goals of modularizing the Java Platform are:[4]

4. "JSR 376: Java Platform Module System." Accessed January 7, 2025. https://jcp.org/en/jsr/detail?id=376.

- **Reliable configuration**—Modularity provides mechanisms for explicitly declaring dependencies between modules in a manner recognized at compile time and execution time. The system can follow these dependencies to determine the subset of all modules required to support your app.
- **Strong encapsulation**—The packages in a module are accessible to other modules only if the module explicitly "`exports`" them. Even then, another module cannot use those packages unless it explicitly states that it "requires" the other module's capabilities. This improves platform security because fewer classes are accessible to potential attackers. Modularity helps you come up with cleaner, more logical designs.
- **Scalable Java Platform**—Before JPMS, the Java Platform was a monolith consisting of massive numbers of packages, making it challenging to develop, maintain and evolve. It couldn't be easily subsetted. The platform is now modularized into dozens of modules. You can create custom runtimes consisting of only modules you need for your apps or the devices you're targeting. For example, if a device does not support GUIs, you could create a runtime that does not include the GUI modules, significantly reducing the runtime's size.
- **Greater platform integrity**—Before Java 9, it was possible to use many classes in the platform that were not meant for use by an app's classes. With strong encapsulation, internal APIs are truly encapsulated and hidden from apps using the platform.
- **Improved performance**—The JVM uses various

optimization techniques to improve program performance.These techniques are more effective when the JVM knows in advance that required types are located only in specific modules.[5]

5. “JSR 376: Java Platform Module System.” Accessed January 7, 2025. https://jcp.org/en/jsr/detail?id=376.

With these goals in mind, many popular open-source Java libraries, including Jackson (Chapter 11) and JavaFX (Chapters 15–17), are now modularized.

Listing the JDK’s Modules

The JDK is divided into modules to support various configurations.[6] The command

6. Mark Reinhold, “JEP 200: The Modular JDK.” Accessed January 7, 2025. https://openjdk.org/jeps/200.

```
java --list-modules
```

uses the **`--list-modules`** option to output the JDK’s set of modules, including **standard modules** that implement the Java SE Specification (“`java.`” names) and JDK-specific modules (“`jdk.`” names):

Click here to view code image

```
java.base@23
java.compiler@23
java.datatransfer@23
java.desktop@23
java.instrument@23
java.logging@23
java.management@23
java.management.rmi@23
java.naming@23
java.net.http@23
java.prefs@23
java.rmi@23
java.scripting@23
java.se@23
java.security.jgss@23
java.security.sasl@23
java.smartcardio@23
java.sql@23
java.sql.rowset@23
java.transaction.xa@23
```

```
java.xml@23
java.xml.crypto@23
jdk.accessibility@23
jdk.attach@23
jdk.charsets@23
jdk.compiler@23
jdk.crypto.cryptoki@23
jdk.crypto.ec@23
jdk.crypto.mscapi@23
jdk.dynalink@23
jdk.editpad@23
jdk.graal.compiler@23
jdk.graal.compiler.management@23
jdk.hotspot.agent@23
jdk.httpserver@23
jdk.incubator.vector@23
jdk.internal.ed@23
jdk.internal.jvmstat@23
jdk.internal.le@23
jdk.internal.md@23
jdk.internal.opt@23
jdk.internal.vm.ci@23
jdk.jartool@23
jdk.javadoc@23
jdk.jcmd@23
jdk.jconsole@23
jdk.jdeps@23
jdk.jdi@23
jdk.jdwp.agent@23
jdk.jfr@23
jdk.jlink@23
jdk.jpackage@23
jdk.jshell@23
jdk.jsobject@23
jdk.jstatd@23
jdk.localedata@23
jdk.management@23
jdk.management.agent@23
jdk.management.jfr@23
jdk.naming.dns@23
jdk.naming.rmi@23
jdk.net@23
jdk.nio.mapmode@23
jdk.sctp@23
jdk.security.auth@23
jdk.security.jgss@23
jdk.unsupported@23
jdk.unsupported.desktop@23
jdk.xml.dom@23
jdk.zipfs@23
```

Each module name is followed by an `@` symbol and a version number. We used Java 23, so each module is followed by `"@23"`. The 12 items in bold are platform-specific and not listed in the documentation's modules overview at

Click here to view code image

```
https://docs.oracle.com/en/java/javase/23/docs/api/index.html
```

We produced the preceding listing with the OpenJDK on Windows 11. The macOS Open-JDK module listing is identical, but `jdk.crypto.mscapi@23` is not in the macOS JDK.

Java Enhancement Proposals (JEPs) and Java Specification Requests (JSRs) of Java Modularity

The preface introduced JEPs and JSRs. The following table contains the Java modularity JEPs and JSRs, many of which we cite these throughout the chapter.

Java Modularity JEPs and JSRs

JEP 200: The Modular JDK (https://openjdk.org/jeps/200)
JEP 201: Modular Source Code (https://openjdk.org/jeps/201)
JEP 220: Modular Run-Time Images (https://openjdk.org/jeps/220)
JEP 260: Encapsulate Most Internal APIs (https://openjdk.org/jeps/260)
JEP 261: Module System (https://openjdk.org/jeps/261)
JEP 275: Modular Java Application Packaging (https://openjdk.org/jeps/275)
JEP 282: `jlink`: The Java Linker (https://openjdk.org/jeps/282)
JEP 396: Strongly Encapsulate JDK Internals by Default (https://openjdk.org/jeps/396)
JEP 403: Strongly Encapsulate JDK Internals (https://openjdk.org/jeps/403)
JSR 376: Java Platform Module System (https://www.jcp.org/en/jsr/detail?id=376)
JSR 379: Java SE 9 (https://www.jcp.org/en/jsr/detail?id=379)

Quick Tour of the Chapter

This chapter introduces key modularity concepts you'll likely use

when building large-scale systems. Some of the key topics presented in this chapter include:

- Module declarations—These specify a module's dependencies (with the `requires` directive), the packages a module makes available to other modules (with the `exports` directive), the services it offers (with the `provides...with` directive), the services it consumes (with the `uses` directive) and to what other modules it allows reflection (with the `open` modifier and the `opens` and `opens...to` directives).
- Module-dependency graphs—We'll create graphs that visualize the dependencies among modules. We'll use the `jdeps` tool to create files that the open-source Graphviz tool (`https://graphviz.org/`) can convert into graphs.
- Module resolver—We'll discuss the steps the JVM's module resolver performs to ensure a module's dependencies are fulfilled.
- `jlink` (the Java linker)—You'll use `jlink` to create smaller custom runtimes then use them to execute apps. In fact, many of this book's command-line apps can be executed on a custom runtime consisting only of the most fundamental JDK module—**`java.base`**—which includes core Java API packages, such as `java.lang`, `java.io` and `java.util`. All modules implicitly depend on `java.base`.
- Reflection—Reflection enables Java code to look inside a class and access and use all its members (including its `private` ones) at execution time.[7] These capabilities can still be used, despite JPMS's strong encapsulation, but only with modules that explicitly allow it. We'll specify that a module allows reflection with an `open` modifier and the `opens` and `opens...to` directives in a module declaration.

  7. *The Java™ Tutorials*, "Trail: The Reflection API." Accessed January 7, 2025. `https://docs.oracle.com/javase/tutorial/reflect/`.

- Migration—The Java platform has been used for almost three decades, so there's enormous amounts of non-modularized legacy code. We'll discuss how the unnamed module and automatic modules can help with migrating legacy code to JPMS. We'll use the `jdeps` tool to determine code dependencies among modules and on internal APIs (which, for the most part, are now strongly encapsulated). Much

legacy code will run without modification, but we explain some potential issues in Section 21.6.

- Services and Service Providers—Substantial software systems that fulfill essential needs can live on for decades. During that time, change is the rule rather than the exception. In Section 9.13, we discussed tight coupling and loose coupling. It's been proven that tight coupling makes it challenging to modify systems. We'll create loosely coupled system components with service-provider interfaces and implementations, and the `ServiceLoader` class. We'll also demonstrate the `uses` and `provides...with` directives in module declarations to indicate that a module uses a service or provides a service implementation, respectively.

We'll present the preceding concepts using live-code examples with meaningful outputs, some code snippets, module graphs containing a module's dependencies, and examples of module-related commands (like `jlink` and `jdeps`) and module-related options for other commands (like `javac`, `java` and `jar`). Some additional example-rich sources include:

- Project Jigsaw: Module System Quick-Start Guide—`https://openjdk.org/projects/jigsaw/quick-start`.
- Mak, Sander, and Paul Bakker. *Java 9 Modularity: Patterns and Practices for Developing Maintainable Applications.* Sebastopol, CA: O'Reilly Media, 2017.
- Parlog, Nicolai. *The Java Module System.* Shelter Island, NY: Manning Publications, 2019.

A Terminology Note

Before Java 9, the Java Runtime Environment (JRE) was a separate software package that included the Java Virtual Machine (JVM) and other software for executing Java programs. With JPMS, the JRE is a proper subset of the Java Development Kit (JDK), which contains all the Java APIs and tools required to create and run Java programs. This chapter uses the terms Java Platform and Java SE Platform synonymously with the JDK.

Generative AI

1 Prompt genAIs, asking how JPMS improves code maintainability

and reusability.

21.2 Module Declarations

As we mentioned, a module must provide a module declaration specifying

- the module's dependencies,
- the packages the module makes available to other modules,
- and more.

The module declaration is defined in a file named **`module-info.java`**. Each module declaration begins with the keyword **`module`**, followed by a unique **module name** and a **module body** enclosed in braces, as in

```
module modulename {
   // module directives
}
```

The module declaration's body can be empty or may contain **module directives**, including `requires`, `exports`, `provides...with`, `uses` and `opens`. As you'll learn in Section 21.3.5, compiling the module declaration creates the **module descriptor**, which is stored in a file named **`module-info.class`** in the module's root folder. Here we briefly introduce each module directive. We'll present actual module declarations beginning in Section 21.3.3.

21.2.1 `requires`

A **`requires` module directive** specifies that this module depends on another module—this relationship is called a **module dependency**. Each module must explicitly state its dependencies. When module A `requires` module B, module A is said to **read** module B and module B **is read by** module A. To specify a dependency on another module, use `requires`, as in:

```
requires modulename;
```

Section 21.3.3 demonstrates a `requires` directive.[8]

8. There is also a `requires static` directive to indicate that a module is required at compile time, but optional at runtime. This is known as an

optional dependency and is beyond this chapter's scope.

21.2.2 `requires transitive`—Implied Readability

To specify a dependency on another module and to ensure that other modules reading your module also read that dependency—known as **implied readability**—use `requires transitive` as in:

Click here to view code image

```
requires transitive modulename;
```

Consider the following directive from the `java.desktop` module declaration:

```
requires transitive java.xml;
```

Any module that reads `java.desktop` also implicitly reads `java.xml`. For example, if a method from the `java.desktop` module returns a type from the `java.xml` module, code in modules that read `java.desktop`, becomes dependent on `java.xml`. Without the `requires transitive` directive in `java.desktop`'s module declaration, such dependent modules will not compile unless they explicitly read `java.xml`.

Java's standard modules must grant implied readability in all cases like the one described here.[9] Also, though a Java standard module may depend on non-standard modules, it must not grant implied readability to them—this ensures that code depending only on Java standard modules is portable across Java implementations.

9. Clark, Iris, and Mark Reinhold. "Java SE 9 (JSR 379)." March 6, 2017. Accessed January 7, 2025. https://cr.openjdk.org/~iris/se/9/java-se-9-pr-spec-01/java-se-9-spec.html#s7.

21.2.3 `exports` and `exports...to`

An **`exports` module directive** specifies one of the module's packages whose `public` types (and their nested `public` and `protected` types) should be accessible to code in all other modules. An **`exports...to` directive** enables you to specify in a comma-separated list precisely which module's or modules' code can access the exported package—known as a **qualified export**. Section 21.4 demonstrates the `exports` directive.

21.2.4 `uses`

A **`uses` module directive** specifies a service used by this module—making the module a **service consumer**. A service is an object of a class that implements the interface or extends the `abstract` class specified in the `uses` directive. Section 21.9.3 demonstrates the `uses` directive.

21.2.5 `provides...with`

A **`provides...with` module directive** specifies that a module provides a service implementation—making the module a **service provider**. The `provides` part of the directive specifies an interface or `abstract` class listed in a module's `uses` directive. The `with` part of the directive specifies the name of the class that `implements` the interface or `extends` the `abstract` class. Section 21.9.6 demonstrates the `provides...with` directive.

21.2.6 `open`, `opens` and `opens...to`[10,11]

10. Buckley, Alex. "JPMS: Modules in the Java Language and JVM." February 23, 2017. Accessed January 7, 2025. https://cr.openjdk.org/~mr/jigsaw/spec/lang-vm.html.

11. Gosling, James, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley, Dan Smith, and Gavin Bierman. "Java Language Specification, Java SE 23 Edition," Section 7.7. August 21, 2024. Accessed January 7, 2025. https://docs.oracle.com/javase/specs/jls/se23/jls23.pdf.

Before Java 9, reflection could be used at runtime to learn about all types in a package and all members of a type—even its `private` members—whether or not you wanted to allow this. So, nothing was truly encapsulated. JPMS provides strong encapsulation. **By default, a type in a module is not accessible to other modules unless it's a `public` type and you export its package**—even then, reflection is still unavailable by default. You expose only the packages you want to expose, and you must explicitly allow reflection, if needed.

Allowing Runtime-Only Access to a Package

An **`opens` module directive** of the form

```
opens package
```

indicates that a specific *package*'s `public` types and their nested `public` and `protected` types are accessible to code in other modules at runtime only. Also, the types in the specified package and all of those types' members are accessible via reflection.

Allowing Runtime-Only Access to a Package By Specific Modules

An **`opens...to` module directive** of the form

Click here to view code image

```
opens package to comma-separated-list-of-modules
```

indicates that a specific *package*'s `public` types and their nested `public` and `protected` types are accessible to code in the listed module(s) at runtime only. Also, the types in the specified package (and all of the types' members) are accessible via reflection to code in the specified modules.

Allowing Runtime-Only Access to All Packages in a Module

If all the packages in a given module should be accessible at runtime and via reflection to all other modules, you may **`open`** the entire module, as in

```
open module modulename {
   // module directives
}
```

Reflection Defaults

A module with runtime reflective access to a package can use the package's `public` types and their nested `public` and `protected` types. However, reflection allows access to all types in the opened package and all members within those types, including `private` members. For more information on using reflection to access all of a type's members, visit

Click here to view code image

```
https://docs.oracle.com/javase/tutorial/reflect/
```

Dependency Injection

Reflection is commonly used with dependency injection. One example of this is an FXML-based JavaFX app, like those presented in Chapters 15–17. When an FXML app loads, the controller object and the GUI components on which it depends are dynamically created as follows:

- First, because the app depends on a controller object that handles the GUI interactions, the `FXMLLoader` injects a controller object into the running app—that is, the `FXMLLoader` uses reflection to locate and load the controller class into memory and to create an object of that class.
- Next, because the controller depends on the GUI components declared in FXML, the `FXMLLoader` creates the GUI-component objects and injects them into the controller object by assigning each to the controller object's corresponding `@FXML` instance variable.

Once this process is complete, the controller can interact with the GUI and respond to its events. We'll use the `opens…to` directive in Section 21.7.1 to allow the `FXMLLoader` to use reflection on a JavaFX app in a custom module.

Generative AI

1 Prompt genAIs for a brief overview of Java reflection and its common use-cases.

2 Prompt genAIs to explain the security implications of using the `opens` directive in a Java module declaration and when to use it.

21.2.7 Restricted Keywords

The keywords `exports`, `module`, `open`, `opens`, `provides`, `requires`, `to`, `transitive`, `uses` and `with` are restricted keywords. They're keywords only in module declarations and may be used as identifiers anywhere else in your code. We mentioned in footnote 8 that there is also a `requires static` module directive. Of course, `static` is a regular keyword.

21.3 Modularized `Welcome` App

In this section, we create a simple `Welcome` app to demonstrate

module fundamentals. We'll:

- create a class that resides in a module,
- provide a module declaration,
- compile the module declaration and the `Welcome` class into a module, and
- run the class containing `main` in that module.

After covering these basics, we'll also demonstrate:

- packaging the `Welcome` app in a modular JAR file and
- running the app from that JAR file.

21.3.1 `Welcome` App's Structure

The app we present in this section consists of two `.java` files:

- `Welcome.java` contains the `Welcome` app class, and
- `module-info.java` contains the module declaration.

By convention, a modularized app has the following folder structure:

Click here to view code image

```
AppFolder
   src
      ModuleNameFolder
         PackageFolders
            JavaSourceCodeFiles
         module-info.java
```

For our `Welcome` app, which will be defined in the package `com.deitel.welcome`, the following diagram presents the folder structure:
The `src` folder stores all of the app's source code. It contains the module's **root folder**, which has the module's name—`com.deitel.welcome` (we'll discuss module naming momentarily). The module's root folder contains nested folders representing the package's directory structure—`com/deitel/welcome`—corresponding to the package `com.deitel.welcome`. This folder contains `Welcome.java`. The module's root folder contains the required

module declaration `module-info.java`.

Module Naming Conventions

Like package names, module names must be unique. To ensure unique package names, you typically begin the name with your organization's Internet domain name in reverse order. Our domain name is `deitel.com`, so we begin our package names with `com.deitel`. Module names also use the reverse-domain-name convention. If multiple modules have the same name at compile time, a compilation error occurs. At runtime, if multiple modules have the same name an exception occurs.

This example uses the same name for the module and its contained package because there is only one package in the module. This is not required but is a common convention. In a modular app, Java maintains the module names separately from package names and any type names in those packages, so duplicate module and package names are allowed.

Modules group related packages. As such, the packages will often have commonality among portions of their names. For example, if a module contains the packages

Click here to view code image

```
com.deitel.sample.firstpackage;
com.deitel.sample.secondpackage;
com.deitel.sample.thirdpackage;
```

you'd typically name the module `com.deitel.sample`—the common portion of the package names. If there's no common portion, choose a name representing the module's purpose. For example, the `java.base` module contains core packages that are considered fundamental to Java apps (such as `java.lang`, `java.io`, `java.time` and `java.util`), and the `java.sql` module contains the packages required for interacting with databases via JDBC (such as `java.sql` and `javax.sql`). These are just two of the many standard modules listed in Section 21.1. The online documentation for each provides a complete list of its exported packages—for the `java.base` module, visit:

Click here to view code image

```
https://docs.oracle.com/en/java/javase/23/docs/api/java.base/module-
   summary.html
```

Listing the `java.base` Module's Contents

You can use the `java` command's **`--describe-modules` option** to display information from the `java.base` module's descriptor, including its list of exported packages, as in:

Click here to view code image

```
java --describe-module java.base
```

Below is a portion of the preceding command's output listing the `java.base` module's packages that are exported so any module can access them. You've used many of these packages in throughout book.

Click here to view code image

```
exports java.io
exports java.lang
exports java.lang.annotation
exports java.lang.classfile
exports java.lang.classfile.attribute
exports java.lang.classfile.components
exports java.lang.classfile.constantpool
exports java.lang.classfile.instruction
exports java.lang.constant
exports java.lang.foreign
exports java.lang.invoke
exports java.lang.module
exports java.lang.ref
exports java.lang.reflect
exports java.lang.runtime
exports java.math
exports java.net
exports java.net.spi
exports java.nio
exports java.nio.channels
exports java.nio.channels.spi
exports java.nio.charset
exports java.nio.charset.spi
exports java.nio.file
exports java.nio.file.attribute
exports java.nio.file.spi
exports java.security
exports java.security.cert
exports java.security.interfaces
exports java.security.spec
exports java.text
exports java.text.spi
```

```
exports java.time
exports java.time.chrono
exports java.time.format
exports java.time.temporal
exports java.time.zone
exports java.util
exports java.util.concurrent
exports java.util.concurrent.atomic
exports java.util.concurrent.locks
exports java.util.function
exports java.util.jar
exports java.util.random
exports java.util.regex
exports java.util.spi
exports java.util.stream
exports java.util.zip
exports javax.crypto
exports javax.crypto.interfaces
exports javax.crypto.spec
exports javax.net
exports javax.net.ssl
exports javax.security.auth
exports javax.security.auth.callback
exports javax.security.auth.login
exports javax.security.auth.spi
exports javax.security.auth.x500
exports javax.security.cert
```

The complete output of the preceding command lists lots of additional `java.base` module information. The following presents a portion of the remaining output with sample lines from each category of information.

Click here to view code image

```
...
uses java.util.spi.CalendarDataProvider
uses java.util.spi.CurrencyNameProvider
...
provides java.nio.file.spi.FileSystemProvider
   with jdk.internal.jrtfs.JrtFileSystemProvider
...
qualified exports com.sun.security.ntlm to java.security.sasl
...
contains jdk.internal
contains jdk.internal.access.foreign
...
```

The `uses` lines like

Click here to view code image

```
uses java.util.spi.CalendarDataProvider
```

indicate there are types in the `java.base` module's packages which use objects that implement various service-provider interfaces. The `provides...with`

Click here to view code image

```
provides java.nio.file.spi.FileSystemProvider
   with jdk.internal.jrtfs.JrtFileSystemProvider
```

indicates that this module's `jdk.internal.jrtfs` package contains a service-provider implementation class named `JrtFileSystemProvider` that implements the service-provider interface `FileSystemProvider` from package `java.nio.file.spi`. Section 21.9 presents a substantial example demonstrating that service-provider interfaces and implementations create loosely coupled system components for systems that are easier to develop, maintain and evolve than tightly coupled systems.

The `qualified exports...to` lines like

Click here to view code image

```
qualified exports com.sun.security.ntlm to java.security.sasl
```

indicate that the `java.base` module `exports` a given package (`com.sun.security.ntlm`) only to a specified module (`java.security.sasl`). The `java.base` module has many of these qualified exports. Packages listed in such exports may be read only by the designated modules in the comma-separated list after the `to` keyword. In the JDK, such qualified exports are used for packages containing JDK internal implementations of types that developers should not use.

The `contains` lines like

Click here to view code image

```
contains com.sun.crypto.provider
```

specify that the module contains packages that are not exported for use in other modules. Note that `contains` is not a directive like `requires` or `exports` that you can use in your modules. Rather, it's information inserted by the compiler to indicate that a module

contains the specified package. The JVM uses this information to improve performance when it loads classes from those packages at runtime.[12]

12. Brian Goetz, e-mail message to authors, March 16, 2017.

21.3.2 Class `Welcome`

Figure 21.1's `Welcome` app simply displays a `String` at the command line. Types that will be placed in modules must be placed into a package using a **`package` statement** (line 3).

Click here to view code image

```
1  // Fig. 21.1: Welcome.java
2  // Welcome class that will be placed in a module
3  package com.deitel.welcome; // all classes in modules must be packaged
4
5  public class Welcome {
6     public static void main(String[] args) {
7        // class System is in package java.lang from the java.base module
8        System.out.println("Welcome to the Java Platform Module System!");
9     }
10 }
```

Fig. 21.1 | `Welcome` class that will be placed in a module.

21.3.3 `module-info.java`

Figure 21.2 is the module declaration for the `com.deitel.welcome` module, which begins with the `module` keyword followed by the module's name and braces that enclose the declaration's body. This module declaration contains a `requires` module directive, indicating that the app depends on types defined in module `java.base`. All modules implicitly depend on `java.base`, so this `requires` directive may be omitted.

Click here to view code image

```
1  // Fig. 21.2: module-info.java
2  // Module declaration for the com.deitel.welcome module
3  module com.deitel.welcome {
4     requires java.base; // implicit in all modules, so can be omitted
5  }
```

Fig. 21.2 | Module declaration for the `com.deitel.welcome` module.

21.3.4 Module-Dependency Graph

The following diagram is the **module-dependency graph** for `com.deitel.welcome`, indicating that the module reads the standard module `java.base`. This dependency is indicated in the diagram with the arrow from `com.deitel.welcome` to `java.base`.

A module-dependency graph illustrates dependencies among **observable modules**— standard modules and any additional modules required by a given app or library module.[13] The graph's nodes are modules, and their dependencies are represented by directed edges (arrows) that connect the nodes. Some edges represent **explicit dependencies** on modules explicitly specified in a module declaration's `requires` clauses (as we'll present in Fig. 21.6). Some edges represent **implicit dependencies** in which one of the required modules, in turn, depends on other modules. The preceding graph was produced using the JDK's `jdeps` tool and the free Graphviz visualization tool. We discuss how to use these tools to create module dependency graphs in Section 21.6.3.

13. Bateman, Alan, Alex Buckley, Jonathan Gibbons and Mark Reinhold. "JEP 261: Module System." Updated August 19, 2024. Accessed January 7, 2025. `https://openjdk.org/jeps/261`.

21.3.5 Compiling a Module

The commands to compile modules and run modularized apps using the JDK's tools are lengthy. Popular IDEs like JetBrains IntelliJ, the Eclipse IDE for Java Developers, Visual Studio Code and NetBeans have build tools that handle these commands for you.

For your convenience, the file `CommandsForCopyPaste.txt` in this chapter's examples folder contains the commands to compile and run the examples. For readability, we split many commands into multiple lines, using platform-specific line-continuation characters. The commands throughout this chapter are presented in macOS format with the backslash (`\`) line-continuation character. Windows users would replace the backslashes with

- carets (`^`) in Command Prompt windows or
- back ticks (`` ` ``) in PowerShell windows.

Compiling the Welcome App

To compile the `Welcome` app's module, open a command window, use the `cd` command to change to this chapter's `WelcomeApp` folder, then enter:

Click here to view code image

```
javac -d mods/com.deitel.welcome \
   src/com.deitel.welcome/module-info.java \
   src/com.deitel.welcome/com/deitel/welcome/Welcome.java
```

The `-d` option indicates that `javac` should place the compiled code in the specified folder.[14] This example's `mods` folder will contain a subfolder named `com.deitel.welcome`, representing the compiled module. The name `mods` is used by convention.

14. The "`d`" is short for "directory," which is another name for a folder.

`Welcome` App's Folder Structure After Compilation

If the code compiles, the `WelcomeApp` folder's `mods` subfolder structure will be:
This is known as the **exploded-module folder** because the folders and `.class` files are not in a JAR (Java archive) file. A JAR is a collection of folders and files compressed into a single file, known as an archive. The exploded module's structure parallels the app's `src` folder structure described in Section 21.3.1. We'll package the app as a JAR shortly. Exploded module folders and modular JAR files (Section 21.3.7) together are **module artifacts**. These can be placed on the **module path**—a list of module artifact locations—when compiling and executing modularized code.[15,16]

15. Reinhold, Mark. "The State of the Module System." March 8, 2016. Accessed January 7, 2025. https://openjdk.org/projects/jigsaw/spec/sotms/#module-artifacts.

16. Bateman, Alan, Alex Buckley, Jonathan Gibbons and Mark Reinhold. "JEP 261: Module System." Updated August 19, 2024. Accessed January 7, 2025. http://openjdk.java.net/jeps/261.

Listing the `com.deitel.welcome` Module's Contents

You can use the `java` command's `--describe-module` option to display information from the `com.deitel.welcome` module descriptor, as in:

Click here to view code image

```
java --module-path mods --describe-module com.deitel.welcome
```

The resulting output:

Click here to view code image

```
com.deitel.welcome file:///Users/pauldeitel/Documents/examples/ch21/
WelcomeApp/mods/com.deitel.welcome/
requires java.base
contains com.deitel.welcome
```

indicates that the module `requires` the standard module `java.base` and `contains` the package `com.deitel.welcome`. Though the module `contains` this package, it is not exported, so other modules cannot use our module's contents. This example's module declaration explicitly required `java.base`, as indicated by the following line in the preceding command's output:

```
requires java.base
```

If the module declaration had implicitly required `java.base`, the previous command's output would instead contain

```
requires java.base mandated
```

where `mandated` indicates the implicit dependence on `java.base`.

Generative AI

1 Prompt genAIs for Java module naming best practices.

2 Prompt genAIs for the instructions on creating and compiling modules with your preferred IDE.

21.3.6 Running an App from a Module's Exploded Folders

To run the `Welcome` app from the module's exploded folders, use the following command (again, from the `WelcomeApp` folder):

Click here to view code image

```
java --module-path mods \
   --module com.deitel.welcome/com.deitel.welcome.Welcome
```

The **`--module-path`** option specifies the module path—in this case, the `mods` folder. The **`--module`** option specifies the module name and the fully qualified class name of the app's entry point—a class containing `main`. The program executes and displays:

Click here to view code image

```
Welcome to the Java Platform Module System!
```

In the preceding command, `--module-path` can be abbreviated as `-p` and `--module` as `-m`.

Generative AI

1 Prompt genAIs for the instructions to run modularized apps using your preferred IDE.

21.3.7 Packaging a Module into a Modular JAR File

The **`jar` command** can package an exploded module folder as a **modular JAR file**[17] containing all the module's files. You specify the JAR file on the module path when you run the app. If a module contains an app's entry point, you can specify that class with the `jar` command's **`--main-class`** option, as in:

17. Bateman, Alan, Alex Buckley, Jonathan Gibbons and Mark Reinhold. "JEP 261: Module System." Updated August 19, 2024. Accessed January 7, 2025. http://openjdk.java.net/jeps/261.

Click here to view code image

```
jar --create --file jars/com.deitel.welcome.jar \
   --main-class com.deitel.welcome.Welcome \
   -C mods/com.deitel.welcome .
```

The options are as follows:

- `--create` specifies that the command should create a new JAR file.
- `--file` specifies the name of the JAR file and is followed by the name—in this case, the file `com.deitel.welcome.jar` will be created in the folder named `jars`, which must exist before running the `jar` command.

- `--main-class` specifies the fully qualified name of the app's entry point—a class that contains a `main` method.
- `-C` specifies which folder contains the files that should be included in the JAR file and is followed by the files to include—the dot (`.`) indicates that all files in the folder should be included.

You can simplify the `--create`, `--file` and `--main-class` options in the preceding command with the shorthand notation `-cfe`, followed by the JAR file name and main class, as in:

Click here to view code image

```
jar -cfe jars/com.deitel.welcome.jar \
   com.deitel.welcome.Welcome \
   -C mods/com.deitel.welcome .
```

Confirming the Contents of a Module's JAR File

After creating the JAR file, you can verify its contents as follows:

Click here to view code image

```
jar --describe-module --file jars/com.deitel.welcome.jar
```

which produces the following output for the `com.deitel.welcome` module:

Click here to view code image

```
com.deitel.welcome jar:file:///Users/pauldeitel/Documents/examples/ch21/
   WelcomeApp/jars/com.deitel.welcome.jar!/module-info.class
requires java.base
contains com.deitel.welcome
main-class com.deitel.welcome.Welcome
```

The output includes:

- the module name (which it obtains from the JAR's `module-info.class` file),
- the modular JAR file's location,
- what modules this module `requires` (`java.base`),
- what this module `contains` (the package `com.deitel.welcome`), and

- the `main-class` that executes when you run the app from this modular JAR file.

Generative AI

1 Prompt genAIs, asking for instructions on how to package modularized Java apps for distribution using in your preferred IDE.

21.3.8 Running the `Welcome` App from a Modular JAR File

Once you place an app in a modular JAR file for which you've specified the entry point, you can execute the app as follows:

Click here to view code image

```
java --module-path jars -m com.deitel.welcome
```

or

Click here to view code image

```
java -p jars -m com.deitel.welcome
```

The program executes and displays:

Click here to view code image

```
Welcome to the Java Platform Module System!
```

If you did not specify the entry point when creating the JAR, you may still run the app by specifying the module name and fully qualified class name, as in:

Click here to view code image

```
java --module-path jars \
   -m com.deitel.welcome/com.deitel.welcome.Welcome
```

or

Click here to view code image

```
java -p jars -m com.deitel.welcome/com.deitel.welcome.Welcome
```

21.3.9 Aside: Classpath vs. Module Path

Before Java 9, the compiler and runtime located types via the classpath—a list of folders and library archive files containing compiled Java classes. In early Java versions, the class-path was defined by a combination of a `CLASSPATH` environment variable, extensions placed in a special JRE subfolder and options provided to the `javac` and `java` commands.

Because types could be loaded from several locations, the order in which they were searched resulted in brittle apps. For example, many years ago, one of the authors installed a Java app from a third-party vendor on his system. The app's installer placed an old version of a Java library into the JRE's extensions folder. Several Java apps on his system depended on a newer library version with additional types and enhanced versions of the library's older types. Because classes in the JRE's extensions folder were loaded before other classes on the classpath,[18] the apps that depended on the newer library version stopped working, failing at runtime with `NoClassDefFoundError`s and `NoSuchMethodError`s—sometimes long after the apps began executing.

18. "Understanding Extension Class Loading." Accessed December 15, 2024. https://docs.oracle.com/javase/tutorial/ext/basics/load.html.

The reliable configuration provided by modules and module descriptors helps eliminate many such runtime classpath problems. A module's dependencies are resolved as an app launches. Section 21.8.4 presents the steps the JRE's module resolver performs at launch time.

The module path may contain only one of each module and every package may be defined in only one module. If two or more modules have the same name or export the same packages, the runtime immediately terminates.

Generative AI

1 Prompt genAIs for a discussion of how JPMS and the module path improve upon the classpath's limitations.

21.4 Creating and Using a Custom Module

To demonstrate a module that depends on a custom module, let's reorganize an earlier, non-modularized example. We'll declare Section 8.7's `Time1` and `Time1Test` classes in separate modules,

then use `Time1` from the module containing `Time1Test`. To make `Time1` available to `Time1Test`, we'll

- export `Time1`'s package from its custom module and
- require the module containing `Time1` from the module containing `Time1Test`.

The following diagram contains the `src` folder structure for the app's two modules.

21.4.1 Exporting a Package for Use in Other Modules

Every class in a module must be declared in a package. So, we added the `package` statement in line 3 (Fig. 21.3) to Fig. 8.10's `Time1.java`.

Click here to view code image

```
1  // Fig. 21.3: Time1.java
2  // Class Time1 that will be placed in a module.
3  package com.deitel.timelibrary;
4
5  public class Time1 {
6     private int hour; // 0 - 23
7     private int minute; // 0 - 59
8     private int second; // 0 - 59
9
10    // set a new time value using universal time; throw an
11    // exception if the hour, minute or second is invalid
12    public void setTime(int hour, int minute, int second) {
13       // validate hour, minute and second
14       if (hour < 0 || hour >= 24 || minute < 0 || minute >= 60 ||
15          second < 0 || second >= 60) {
16          throw new IllegalArgumentException(
17             "hour, minute and/or second was out of range");
18       }
19
20       this.hour = hour;
21       this.minute = minute;
22       this.second = second;
23    }
24
25    // convert to String in universal-time format (HH:MM:SS)
26    public String toUniversalString() {
27       return String.format("%02d:%02d:%02d", hour, minute, second);
```

```
28      }
29
30      // convert to String in standard-time format (H:MM:SS AM or PM)
31      public String toString() {
32         return String.format("%d:%02d:%02d %s",
33            ((hour == 0 || hour == 12) ? 12 : hour % 12),
34            minute, second, (hour < 12 ? "AM" : "PM"));
35      }
36   }
```

Fig. 21.3 | Class `Time1` that will be placed in a module.

`com.deitel.timelibrary` Module Declaration

In Fig. 21.4's module declaration, line 4 `exports` the module's `com.deitel.timelibrary` package. This package's `public` class `Time1` (and other public classes, if any) can be used by any module that reads the `com.deitel.timelibrary` module, as long as the module can be found on the module path (Section 21.4.3).

Click here to view code image

```
1  // Fig. 21.4: module-info.java
2  // Module declaration for the com.deitel.timelibrary module
3  module com.deitel.timelibrary {
4     exports com.deitel.timelibrary; // package available to other modules
5  }
```

Fig. 21.4 | Module declaration for the `com.deitel.timelibrary` module.

21.4.2 Using a Class from a Package in Another Module

The app's entry point—class `Time1Test` (originally declared in Fig. 8.11)—also must be packaged for placement in a module (line 3 of Fig. 21.5). In addition, class `Time1Test` manipulates an object of class `Time1`, which is declared in a different package. So, line 5 imports the `com.deitel.timelibrary` package's `Time1` class.

Click here to view code image

```
1  // Fig. 21.5: Time1Test.java
2  // Time1 object used in an app.
3  package com.deitel.timetest;
```

```
4
5  import com.deitel.timelibrary.Time1;
6
7  public class Time1Test {
8     public static void main(String[] args) {
9        // create and initialize a Time1 object
10       var time = new Time1(); // invokes Time1 constructor
11
12       // output string representations of the time
13       displayTime("After time object is created", time);
14       System.out.println();
15
16       // change time and output updated time
17       time.setTime(13, 27, 6);
18       displayTime("After calling setTime", time);
19       System.out.println();
20
21       // attempt to set time with invalid values
22       try {
23          time.setTime(99, 99, 99); // all values out of range
24       }
25       catch (IllegalArgumentException e) {
26          System.out.printf("Exception: %s%n%n", e.getMessage());
27       }
28
29       // display time after attempt to set invalid values
30       displayTime("After calling setTime with invalid values", time);
31    }
32
33    // displays a Time1 object in 24-hour and 12-hour formats
34    private static void displayTime(String header, Time1 t) {
35       System.out.printf("%s%nUniversal time: %s%nStandard time: %s%n",
36          header, t.toUniversalString(), t.toString());
37    }
38 }
```

Fig. 21.5 | `Time1` object used in an app.

`com.deitel.timetest` Module Declaration

`Time1` is located in a package of the `com.deitel.timelibrary` module, so the module containing class `Time1Test` (`com.deitel.timetest`) must declare its dependency on the `com.deitel.timelibrary` module. The module declaration (Fig. 21.6) indicates this dependency with line 4's `requires` directive. Due to JPMS's strong encapsulation and reliable configuration features, this app needs both the `exports` directive in Fig. 21.4 and the `requires` directive in Fig. 21.6 for class `Time1Test` to import

and use class `Time1`.

Click here to view code image

```
// Fig. 21.6: module-info.java
// Module declaration for the com.deitel.timetest module
module com.deitel.timetest {
   requires com.deitel.timelibrary;
}
```

Fig. 21.6 | Module declaration for the `com.deitel.timetest` module.

`com.deitel.timetest` Module-dependency Graph

The following `Time1Test` app module-dependency graph indicates that the module `com.deitel.timetest` reads (depend on) the custom module `com.deitel.timelibrary` and the standard module `java.base`.

21.4.3 Compiling and Running the Example

You must compile both modules before running this app. The `com.deitel.timelibrary` module must be compiled first because `com.deitel.timetest` depends on it. IDEs and other build tools (like Ant, Gradle and Maven) typically can deal with order-of-compilation issues like this for you.

Compiling Module `com.deitel.timelibrary`

To compile the `com.deitel.timelibrary` module, open a command window, use the `cd` command to change to this chapter's `TimeApp` folder on your system, then enter:

Click here to view code image

```
javac -d mods/com.deitel.timelibrary \
   src/com.deitel.timelibrary/module-info.java \
   src/com.deitel.timelibrary/com/deitel/timelibrary/Time1.java
```

Compiling Module `com.deitel.timetest`

Next, to compile the `com.deitel.timetest` module, enter:

Click here to view code image

```
javac --module-path mods -d mods/com.deitel.timetest \
   src/com.deitel.timetest/module-info.java \
   src/com.deitel.timetest/com/deitel/timetest/Time1Test.java
```

Here, we added the option `--module-path` to indicate that the `mods` folder contains modules on which the `com.deitel.timetest` module depends—in this case, we previously compiled the `com.deitel.timelibrary` module into the `mods` folder.

Running the Example

Finally, to run this example, enter:

Click here to view code image

```
java --module-path mods \
   -m com.deitel.timetest/com.deitel.timetest.Time1Test
```

In this command:

- the option `--module-path` indicates where the app's modules are located, and
- the option `-m` specifies which class should be used as the app's entry point—that is, a class containing the `main` method that the JVM calls to launch the app.

For the `main` class, note that you must specify its module name followed by a slash and its fully qualified class name because the class is now in a package contained in a module. The program's output is as follows:

Click here to view code image

```
After time object is created
Universal time: 00:00:00
Standard time: 12:00:00 AM

After calling setTime
Universal time: 13:27:06
Standard time: 1:27:06 PM

Exception: hour, minute and/or second was out of range

After calling setTime with invalid values
Universal time: 13:27:06
Standard time: 1:27:06 PM
```

21.4.4 Packaging the App into Modular JAR Files

In this section, we'll package each app into a modular JAR file then run the app. To package `com.deitel.timelibrary` into a modular JAR file, enter:

Click here to view code image

```
jar --create -f jars/com.deitel.timelibrary.jar \
   -C mods/com.deitel.timelibrary .
```

To package `com.deitel.timetest` into a modular JAR file, enter:

Click here to view code image

```
jar --create -f jars/com.deitel.timetest.jar \
   --main-class com.deitel.timetest.Time1Test \
   -C mods/com.deitel.timetest .
```

Running the App from a Modular JAR File

Once you place an app in a modular JAR file for which you've specified the `main` class, you can execute the app as follows:

Click here to view code image

```
java --module-path jars -m com.deitel.timetest
```

The program executes and displays the same output as in Section 21.4.3.

21.4.5 Strong Encapsulation and Accessibility

Before modules, you could use any `public` class you imported into your code. Whether you could access the class's members was determined by how they were declared—`public`, `protected`, package access or `private` (as described in Chapter 8). Due to modules' **strong encapsulation**, `public` types in a module are no longer accessible to your code by default. So, `public` does not necessarily mean "available to all":

- If a module exports a package, the `public` types in that package are accessible to any module that reads the

package's module.

- If a module exports a package to a specific module (via `exports...to`), the `public` types in that package are accessible only to the specific module and only if that module reads the package's module.
- If a module does not export a package, the `public` types in that package are accessible only within the module itself.

Once you have access to a type in another module, the normal rules of `public`, `protected`, package access and `private` apply.

Compilation Error When Attempting to Use an Inaccessible Type

The project `TimeAppMissingExports` in this chapter's `ExamplesShowingErrors` folder demonstrates that explicitly named modules have strong encapsulation and do not export packages unless you explicitly list them in `exports` directives. In this project, we removed the `exports` directive from the `com.deitel.timelibrary`'s module declaration then recompiled the module. Next, we compiled the `com.deitel.timelibrary` module then tried to recompile the `com.deitel.timetest` module. The compiler produced the following error message, which indicates that the package `com.deitel.timelibrary` is not exported and thus is inaccessible:

Click here to view code image

```
src/com.deitel.timetest/com/deitel/timetest/Time1Test.java:5: error: package
com.deitel.timelibrary is not visible
import com.deitel.timelibrary.Time1;
          ^
   (package com.deitel.timelibrary is declared in module com.deitel.timeli-
brary, which does not export it)
1 error
```

When a `requires` dependency is not fulfilled by an `exports` clause in another module a compilation error occurs.

Generative AI

1 Prompt genAIs for recommendations on when a module should use `exports` vs. `exports...to`.

21.5 Module-Dependency Graphs: A Deeper Look

Let's continue our discussion of module graphs and present the errors that occur if a module directly or indirectly `requires` itself—known as a cycle.

21.5.1 `java.sql`

Consider the following `modulegraphtest` module that depends on the `java.sql` module:

```
module modulegraphtest {
   requires java.sql;
}
```

The following diagram is the module-dependency graph for `modulegraphtest`:
The diagram illustrates that `modulegraphtest` depends directly on modules `java.sql` and `java.base`. Modules `java.logging`, `java.transaction.xa` and `java.xml` are included in the graph because `java.sql` depends on them—thus, `modulegraphtest` also depends on them indirectly. The `java.sql` module's `module-info.java` declares these modules with the following `requires transitive` statements:

Click here to view code image

```
requires transitive java.logging;
requires transitive java.transaction.xa;
requires transitive java.xml;
```

So, modules that depend on `java.sql` also have access to these modules. Of course, all the modules in the graph point to `java.base` because it's required by all Java modules.

21.5.2 `java.se`

Figure 21.7 presents the significantly more complex **`java.se`** module's dependency graph. This **aggregator module** specifies via `requires transitive` all the modules necessary to support Java Standard Edition (SE) apps. We rotated the graph 90° for readability.

Fig. 21.7 | `java.se` module-dependency graph.

21.5.3 Error: Module Graph with a Cycle

A module is not allowed to directly or indirectly reference itself. Doing so would result in a cycle when computing the module's dependency graph. A compilation error occurs if a module graph contains a cycle.

A Module That (Incorrectly) Requires Itself

Consider the following module declaration in which the module requires itself:

```
module mymodule {
   requires mymodule;
}
```

If you were to compile this declaration, the following error would occur, indicating a cycle in the `module`'s dependencies:

Click here to view code image

```
module-info.java:2: error: cyclic dependence involving mymodule
   requires mymodule;
            ^
1 error
```

Two Modules That (Incorrectly) Require One Another

Similarly, consider the project `CircularDependency` containing two modules—`module1` and `module2`—with the structure in the following diagram:
If the module declarations for these two modules indicate that each module `requires` the other, as in

```
module module1 {
   exports package1;
   requires module2;
}
```

and

```
module module2 {
```

```
   exports package2;
   requires module1;
}
```

then, when you compile these modules

Click here to view code image

```
javac --module-source-path src \
   --module-path mods -d mods \
   src/module1/module-info.java \
   src/module1/package1/Class1.java \
   src/module2/module-info.java \
   src/module2/package2/Class2.java
```

the compiler again issues an error indicating a cycle in the module dependencies:

Click here to view code image

```
src/module1/module-info.java:9: error: cyclic dependence involving module2
   requires module2;
            ^
src/module2/module-info.java:9: error: cyclic dependence involving module1
   requires module1;
            ^
2 errors
```

Modules in a Cycle Are Really "One Thing"

Ultimately, all the modules in a cycle are really one module—not separate modules.[19] When we were writing this chapter for *Java How to Program, 11/e*, a friend who worked for a large organization indicated that the organization had multiple large pre-Java-9 JAR files. Initially they thought they'd make each JAR a separate module. Their JARs were so interdependent that they combined them into a single module. This kind of interdependency is what leads to cycles in your design. Ideally, when you modularize a previously monolithic system, you want to break that system into separate modules that are easier to maintain and secure. This can pose significant refactoring challenges in large code bases.

19. Alex Buckley, e-mail message to authors, March 24, 2017.

Generative AI

1 Prompt genAIs for recommendations on how to avoid cyclic dependencies among modules.

21.6 Migrating Code to Modules

Many non-modularized apps will run using the modularized JDK. In fact, most programs presented in this book were not modularized and ran without issue. Technically, all programs are compiled and executed using the module system. Java strongly encapsulates types that are not exported by modules, so it's possible some older apps will fail to compile in Java 9 and beyond because types that were accessible to them before Java 9 no longer are. For example, there are pre-Java-9 **internal APIs** that were meant only for use in the JDK but were used by developers in their apps and libraries. Most of these internal APIs have been replaced with proper publicly accessible capabilities, so the internal APIs are no longer exported and thus are inaccessible.[20]

20. Reinhold, Mark. "JEP 260: Encapsulate Most Internal APIs." Updated April 28, 2024. Accessed January 7, 2025. https://openjdk.org/jeps/260.

Strong Encapsulation Is Now the Default

As we've mentioned, JPMS enables strong encapsulation. Code that is not exported cannot be accessed by other modules. Java versions 9–15 allowed some flexibility with respect to internal APIs to give developers time to migrate their code to JPMS. Since Java 16 and Java 17, strong encapsulation is the default.[21] If you have older code that uses such internal APIs directly or indirectly, it will fail to compile.

21. Buckley, Alex and Mark Reinhold, "JEP 396: Strongly Encapsulate JDK Internals by Default." Updated April 28, 2024. Accessed December 17, 2024. https://openjdk.org/jeps/396.

Checking Your Code's Dependencies

You can use the `jdeps` tool (Section 21.6.3) to check your code's dependencies. The `jdeps` option `--jdk-internals` specifically identifies uses of JDK internal APIs in code. For each internal API that `jdeps` locates, it suggests an appropriate replacement.

21.6.1 Unnamed Module

Technically, Java requires all code to be placed in modules. When you execute code that's not in a module, the code is loaded from the classpath and placed in the **unnamed module**. This is why we can run some non-modularized code in the modularized JDK but without the benefits of strong encapsulation and explicit module dependencies.

The unnamed module:

- implicitly `exports` all of its packages, and
- implicitly reads all other modules.

There's no way to refer to an unnamed module in a `requires` directive from a named module, so a named module cannot depend on an unnamed module.

21.6.2 Automatic Modules

There are enormous numbers of preexisting libraries that you can use in your apps. Some are modularized and some are not. To facilitate migration, you can add any library's JAR file to an app's module path then use the packages in that JAR. When you do, the JAR file implicitly becomes an **automatic module** and can be specified in a `module` declaration's `requires` directives. An automatic module:

- implicitly `exports` all of its packages—so any module that reads the automatic module (including the unnamed module) has access to the `public` types in the automatic module's packages.
- implicitly reads (`requires`) all other modules, including other automatic modules and the unnamed module—so an automatic module has access to all the `public` types exposed by the system's other modules.

Automatic Module Names Derived from JAR File Names

An automatic module's name is derived from its file name. If the JAR's file name minus the `.jar` extension is a valid Java identifier, it becomes its automatic module name. Otherwise:

- Characters that are not allowed in Java identifiers are replaced with dots (`.`).
- If there are multiple consecutive dots, they are replaced with a single dot.
- If there are dots at the beginning or end of the remaining name, they're removed.

21.6.3 `jdeps` Tool for Java Dependency Analysis

Another tool to help you prepare to modularize your code is the **`jdeps` command**. It enables you determine a type's class and package dependencies. Given that 28.8% of Java developers were still using Java 8 in 2024,[22] a key use of `jdeps` is to help those developers locate dependencies on internal APIs that are now strongly encapsulated. To determine whether a class has any such dependencies, use the following command on your compiled class:

22. New Relic, "2024 State of the Java Ecosystem," April 2024. Accessed December 15, 2024. https://newrelic.com/resources/report/2024-state-of-the-java-ecosystem.

Click here to view code image

```
jdeps --jdk-internals YourClassName.class
```

or if you have many compiled classes in a JAR file, use:

Click here to view code image

```
jdeps --jdk-internals YourJARName.jar
```

If this command does not produce output, then your class or set of classes does not depend on JDK internal APIs that are no longer accessible.

Determining the Modules You Need

A key aspect of modularization is the ability to create more compact Java runtimes that provide only the modules your app requires. When you're preparing to create custom runtimes, you can use `jdeps` to discover your app's module dependencies so you know which ones to include in your custom runtime. For example, this chapter's `Welcome` app depends only on `java.base`. We can confirm

that by executing the following command from the `WelcomeApp` folder, which checks the `com.deitel.welcome` module's dependencies:

Click here to view code image

```
jdeps --module-path jars -m com.deitel.welcome
```

This produces the following output containing the packages and modules the app uses— we removed some horizontal spacing in this command's output to make it more readable:

Click here to view code image

```
com.deitel.welcome
 [file:///Users/pauldeitel/Documents/examples/ch21/WelcomeApp/jars/
   com.deitel.welcome.jar]
   requires java.base (@23)
com.deitel.welcome -> java.base
   com.deitel.welcome -> java.io        java.base
   com.deitel.welcome -> java.lang      java.base
```

The output indicates that our module `com.deitel.welcome` depends on the `java.base` module, and that our module specifically uses types from the `java.base` module's `java.io` and `java.lang` packages. The preceding command may also be written as

Click here to view code image

```
jdeps jars/com.deitel.welcome.jar
```

In addition, you can use `jdeps` on a specific `.class` file, as in:

Click here to view code image

```
jdeps mods/com.deitel.welcome/com/deitel/welcome/Welcome.class
```

which produces

Click here to view code image

```
Welcome.class -> java.base
   com.deitel.welcome -> java.io        java.base
   com.deitel.welcome -> java.lang      java.base
```

Verbose `jdeps` Output

If you'd like more details, you can specify the `-v` (verbose) option as in:

Click here to view code image

```
jdeps -v jars/com.deitel.welcome.jar
```

which produces:

Click here to view code image

```
com.deitel.welcome
 [file:///Users/pauldeitel/Documents/examples/ch21/WelcomeApp/jars/
   com.deitel.welcome.jar]
   requires java.base (@23)
com.deitel.welcome -> java.base
   com.deitel.welcome.Welcome -> java.io.PrintStream      java.base
   com.deitel.welcome.Welcome -> java.lang.Object         java.base
   com.deitel.welcome.Welcome -> java.lang.String         java.base
   com.deitel.welcome.Welcome -> java.lang.System         java.base
```

This lists precisely which packages, types and modules the app uses. Knowing that the app requires only `java.base`, we can then use `jlink` to create a custom runtime containing only that module, which we'll do in Section 21.8.

Using `jdeps` to Produce DOT Files for Graphing Tools

You can use graphing tools—such as Graphviz (`https://www.graphviz.org/`)—to produce module-dependency graphs using the DOT graph description language,[23] which specifies a graph's nodes and edges. The `jdeps` tool can create DOT (`.dot`) files with the `--dot-output` option, as in:

23. "DOT (graph description language)." Wikipedia, Wikimedia Foundation. Last modified November 12, 2024. Accessed December 15, 2024. `https://en.wikipedia.org/wiki/DOT_(graph_description_language)`.

Click here to view code image

```
jdeps --dot-output . jars/com.deitel.welcome.jar
```

This produces two `.dot` files[24] in the current folder (`.`):

24. The `.dot` extension is also used by Microsoft Word document templates. On systems with Microsoft Word installed, open the `jdeps`-produced `.dot` files directly from a text editor.

- `summary.dot`—the description of module `com.deitel.welcome`'s dependencies.
- `com.deitel.welcome.dot`—the description of module `com.deitel.welcome`'s specific package dependencies.

After installing Graphviz, we executed the following command to visualize `summary.dot`:

Click here to view code image

```
dot -Tpng summary.dot -Gdpi=600 -Nfontname="Open Sans" \
   -Nshape=box -o WelcomeApp.png
```

- `-Tpng` specifies the image should be in the popular PNG image format commonly used on the web.
- `-Gdpi` specifies the image resolution—600 dots-per-inch.
- `-Nfontname` option enables you to choose a font for the nodes' text.
- `-Nshape` specifies the nodes' shape—`box` produces rectangles rather than the default ovals.
- `-o` is followed by the image's filename.

The diagram below illustrates the `com.deitel.welcome` module's dependencies:

Additional `jdeps` Options

For a complete list of `jdeps` options, visit

Click here to view code image

```
https://docs.oracle.com/en/java/javase/23/docs/specs/man/jdeps.html
```

Generative AI

1 Prompt genAIs for an overview of the strategies and best practices to migrate non-modularized Java code to JPMS.

21.7 Resources in Modules

When the types in a module require resources—such as images, videos, XML documents and more—those resources should be packaged with the module to ensure that they're available when the module's types are used at execution time. This is known as **resource encapsulation**.[25] In this section, we'll migrate our non-modularized JavaFX `VideoPlayer` example from Section 17.6 into a module that also encapsulates the app's resources—the FXML file that describes the GUI and its video file that will be loaded and played at execution time. By convention, resources typically are placed in a folder named `res`.

25. Mark Reinhold, "Java Platform Module System Requirements." April 1, 2015. Accessed January 7, 2025. https://openjdk.org/projects/jigsaw/spec/reqs/#resource-encapsulation.

Recall that the original `VideoPlayer` example consisted of the following files all in Chapter 17's `VideoPlayer` folder:

- `VideoPlayer.xml`—The FXML file that describes the app's GUI.
- `VideoPlayer.java`—The `Application` subclass that begins the app's execution.
- `VideoPlayerController.java`—The controller class that responds to the GUIs events and loads the video.
- `A1Launch~medium.mp4`—The NASA video[26] that the app loads and plays.

  26. For NASA's terms of use, visit https://www.nasa.gov/nasa-brand-center/images-and-media/

- `controlsfx-11.2.1.jar`—The ControlsFX library containing the dialog class `ExceptionDialog`. We display an `ExceptionDialog` if the `MediaPlayer` encounters any errors.

Reorganizing for Modularization

For this example, we reorganized the files into the following folder structure to support modules.
Notice the following about the structure:

- The files `VideoPlayer.fxml` and `A1Launch~medium.mp4`, which are not Java source code files, are located in the module directory's `res` folder. These files will be read from the module's `res` folder when the app executes.
- As required for modularization, we placed the `VideoPlayer`

and `VideoPlayer-Controller` classes in a package—the folder structure `com/deitel/videoplayer` corresponds to the package `com.deitel.videoplayer`.

- As required, we created a `module-info.java` file in the module's root folder.

In addition, we placed `controlsfx-11.2.1.jar` in the `VideoPlayer` folder's `mods` subfolder.

Code Changes for Modularization

We made several code changes to the `VideoPlayer` application for modularization. In `VideoPlayer.fxml`, we modified the controller class's name to use its fully qualified name

Click here to view code image

```
com.deitel.videoplayer.VideoPlayerController
```

so that the `FXMLLoader` can find the controller class. In `VideoPlayer.java`, we changed `"VideoPlayer.fxml"` to `"/res/VideoPlayer.fxml"` to indicate that the FXML file is located in the module's `res` subfolder. We also added the `package` statement

Click here to view code image

```
package com.deitel.videoplayer;
```

In `VideoPlayerController.java`, we changed `"A1Launch~medium.mp4"` to

```
"/res/A1Launch~medium.mp4"
```

to indicate that the video file is located in the module's `res` folder. We also added the `package` statement

Click here to view code image

```
package com.deitel.videoplayer;
```

The rest of the code is identical to what we presented in Section 17.6.

21.7.1 Requiring Multiple Modules

Module `com.deitel.videoplayer`'s declaration (Fig. 21.8) indicates

that it requires `javafx.controls`, `javafx.fxml`, `javafx.media` and `org.controlsfx.controls`. The module `exports` package `com.deitel.videoplayer` (line 9), because class `VideoPlayer-Controller` is used by class `FXMLLoader` (module `javafx.fxml`) when it creates the controller object and the app's GUI.

Click here to view code image

```
1  // Fig. 21.8: module-info.java
2  // Module declaration for the com.deitel.videoplayer module
3  module com.deitel.videoplayer {
4     requires javafx.controls;
5     requires javafx.fxml;
6     requires javafx.media;
7     requires org.controlsfx.controls;
8
9     exports com.deitel.videoplayer;
10    opens com.deitel.videoplayer to javafx.fxml;
11 }
```

Fig. 21.8 | Module declaration for the `com.deitel.videoplayer` module.

21.7.2 Opening a Module for Reflection

In Fig. 21.8, the `opens...to` directive (line 10) indicates that the accessible types in the package `com.deitel.videoplayer` should be available via reflection at runtime to types in the `javafx.fxml` module. As we discussed in Section 21.2.6, this enables the `FXMLLoader` to locate and load class `VideoPlayerController`. The `FXMLLoader` creates a `VideoPlayer-Controller` object and injects into it references to the GUIs components created from the app's FXML file. For one module to open a package to another module, that package must first be exported (possibly as a qualified export using `exports...to`).

21.7.3 Compiling the Module

To compile the `com.deitel.videoplayer` module, enter:

Click here to view code image

```
javac --module-path "$PATH_TO_FX":mods \
   -d mods/com.deitel.videoplayer \
   src/com.deitel.videoplayer/module-info.java \
```

```
src/com.deitel.videoplayer/com/deitel/videoplayer/*.java
```

We included the `--module-path` option to specify where the JavaFX modules are (`"$PATH_TO_FX"`) and where `controlsfx-11.2.1.jar` is (`mods`).

Copying the Resource Files into the Module

Though some IDEs and build tools will automatically put the module's resources into the compiled module, the preceding `javac` command does not. Once you've compiled the module, copy the `res` folder from this project's `src/com.deitel.videoplayer` folder into the `mods/com.deitel.videoplayer` folder.

21.7.4 Running a Modularized App

To execute class `VideoPlayer` from the `com.deitel.videoplayer` module, enter:

Click here to view code image

```
java --module-path "$PATH_TO_FX":mods \
   -m com.deitel.videoplayer/com.deitel.videoplayer.VideoPlayer
```

Here is the app executing on macOS:

21.7.5 Module-Dependency Graph

The following diagram is the `com.deitel.videoplayer` module-dependency graph:
We produced this diagram using the following `jdeps` command to create `summary.dot`— the `--recursive` option indicates that jdeps should analyze all the runtime dependencies of the specified module:

Click here to view code image

```
jdeps --module-path "$PATH_TO_FX":mods --dot-output . \
   --recursive --module com.deitel.videoplayer
```

We then used Graphviz to visualize it:

Click here to view code image

```
dot -Tpng summary.dot -Gdpi=600 -Nfontname="Open Sans" \
   -Nshape=box -o VideoPlayer.png
```

See the file `CommandsForCopyPaste.txt` for the Windows version of this command.

Generative AI

1 Prompt genAIs for examples of the resource types you can encapsulate in a Java module.

2 Prompt genAIs for the benefits of encapsulating resources in a Java module.

21.8 Creating Custom Runtimes with `jlink`

Before JPMS, every Java SE program required the entire Java runtime to execute, even if the program used only a small portion of the Java platform's capabilities. The **`jlink` command** enables you to create custom runtime images.[27] In a custom runtime, you can include only what's necessary for a given app or set of apps to execute. This is particularly useful for executing large numbers of Java runtimes in cloud-based architectures, where you pay for all the storage, memory and processor resources you use. It's also useful for smaller capacity devices. If your app requires only packages from the `java.base` module, you can create a runtime containing just that module. In fact, such a runtime can execute most of this book's text-only, command-line examples.

27. Denise, Jean-Francois. "JEP 282: `jlink`: The Java Linker." Updated May 19, 2017. Accessed January 7, 2025. https://openjdk.org/jeps/282.

21.8.1 Custom Runtime Containing Only `java.base`

For this example, change to the `WelcomeApp` folder—after creating the custom runtime, you'll execute the `Welcome` app using it. The following command creates a runtime containing only the module `java.base`:

Click here to view code image

```
jlink --module-path "$JAVA_HOME/jmods" --add-modules java.base \
   --output javabaseruntime
```

The commands options are as follows:

- `--module-path` specifies one or more folders in which to locate modules to include in the runtime—in this case, the JDK's `jmods` folder, which contains the JDK's modules. Place `"$JAVA_HOME"` in quotes in case the path to the JDK on your system contains spaces.
- `--add-modules` specifies which modules to include in the runtime—in this case, just `java.base`.
- `--output` specifies the folder (`javabaseruntime`) in which the runtime's contents are placed. This folder will be a subfolder of the one in which you execute the preceding command (unless you specify additional path information). If the folder already exists, an error occurs.

This runtime can execute an app that depends only on types from the packages in module `java.base`, including many of this book's command-line apps.

Note Regarding the `JAVA_HOME` Environment Variable

The `JAVA_HOME` environment variable must refer to the JDK's installation folder on your system. If you installed the JDK using the instructions in our Before You Begin section that follows the preface, the installer set up the `JAVA_HOME` environment variable for you. You may need to set up this environment variable yourself if you used a different Java distribution.

Executing the `Welcome` App Using This Custom Runtime

To run the app with the custom runtime, execute:

Click here to view code image

```
javabaseruntime/bin/java --module-path mods \
   --module com.deitel.welcome/com.deitel.welcome.Welcome
```

The program executes and displays:

Click here to view code image

```
Welcome to the Java Platform Module System!
```

Listing the Modules in a Custom Runtime

Early in this chapter, we listed all the modules in the JDK with the command

```
java --list-modules
```

Once you have a custom runtime, you can use the `java` command from the custom runtime's `bin` folder to confirm the modules it includes, as in:

Click here to view code image

```
javabaseruntime/bin/java --list-modules
```

When executing the custom runtime's `java` command, use `/` to separate folder names on macOS and `\` to separate the folder names on Windows. The preceding command produces the following output:

```
java.base@23
```

Similarly the following command creates a custom runtime containing only the module `java.sql` and any other modules on which it depends:

Click here to view code image

```
jlink --module-path "$JAVA_HOME/jmods" \
   --add-modules java.sql --output javasqlruntime
```

For this custom runtime, running

Click here to view code image

```
javasqlruntime/bin/java --list-modules
```

(again, use forward slashes on macOS and Linux) produces the following output:

```
java.base@23
java.logging@23
java.sql@23
java.transaction.xa@23
java.xml@23
```

21.8.2 Creating a Custom Runtime for the `Welcome` App

To create a custom runtime containing only the modules `com.deitel.welcome` and its dependencies (in this case, `java.base`), change to the `WelcomeApp` folder, then use the following command:

Click here to view code image

```
jlink --module-path jars:"$JAVA_HOME/jmods" \
   --add-modules com.deitel.welcome --output welcomeruntime
```

This creates a custom runtime in the folder `welcomeruntime`. The preceding command specifies multiple folders—`jars` and `"$JAVA_HOME/jmods"`. On macOS, the path-separator character for lists of folders is a colon (`:`). The Windows version of this command is located in the file `CommandsForCopyPaste.txt` with this chapter's examples. For the list of modules included in the custom runtime, on macOS use:

Click here to view code image

```
welcomeruntime/bin/java --list-modules
```

(use backslashes on Windows), which produces the following list of modules:

```
com.deitel.welcome
java.base@23
```

Generative AI

1 Prompt genAIs with the code and FXML file for the `VideoPlayer` app in Section 21.7. Ask them to create the `jlink` command for your operating system to build a custom runtime with which you could execute the app.

21.8.3 Executing the `Welcome` App Using a Custom Runtime

To run the app with the custom runtime, use the following command (use backslashes on Windows):

Click here to view code image

```
welcomeruntime/bin/java -m com.deitel.welcome
```

This executes the program and displays:

Click here to view code image

```
Welcome to the Java Platform Module System!
```

21.8.4 Using the Module Resolver on a Custom Runtime

When you run a modularized app, the JVM's **module resolver** determines which modules are required at execution time and ensures their dependencies are satisfied—this is known as the **transitive closure** of those modules. To locate modules, the module resolver looks at the **observable modules**—those built into the runtime (like `java.base`) and those located on the module path. The runtime throws a `java.lang.module.FindException` if a required module cannot be found.

For a given app and runtime, you can view the steps the module resolver follows to determine module dependencies and ensure that the required modules are available to the program. To do so, include the **`--show-module-resolution` option**[28] in the `java` command, as in the following command (use backslashes on Windows):

28. Bateman, Alan, Alex Buckley, Jonathan Gibbons and Mark Reinhold. "JEP 261: Module System." Updated August 19, 2024. Accessed January 7, 2025. https://openjdk.org/jeps/261.

Click here to view code image

```
welcomeruntime/bin/java --show-module-resolution -m com.deitel.welcome
```

This uses the custom `welcomeruntime`'s `java` command to display the resolver's steps for locating modules, followed by the program's output:

Click here to view code image

```
root com.deitel.welcome jrt:/com.deitel.welcome
Welcome to the Java Platform Module System!
```

In the `Welcome` app's module-resolution process, the resolver first locates the app's **initial module**—`com.deitel.welcome`—containing the app's entry point. The resolver refers to this as the `root` module—the root node in the module-dependency graph. Then, the resolver looks for other module dependencies. The `Welcome` app's only other dependency is the `java.base` module. Since all modules depend on this, it's not listed in the output.

Adding the `--show-module-resolution` option to the `java` command for the `Video-Player` app (Section 21.7) produces a substantial output, a portion of which is shown below:

Click here to view code image

```
root com.deitel.videoplayer file:///Users/pauldeitel/Documents/examples/ch21/
   VideoPlayer/mods/com.deitel.videoplayer/
com.deitel.videoplayer requires javafx.media file:///Users/pauldeitel/
   Downloads/javafx-sdk-23.0.1/lib/javafx.media.jar
com.deitel.videoplayer requires javafx.controls file:///Users/pauldeitel/
   Downloads/javafx-sdk-23.0.1/lib/javafx.controls.jar
com.deitel.videoplayer requires javafx.fxml file:///Users/pauldeitel/
   Downloads/javafx-sdk-23.0.1/lib/javafx.fxml.jar
com.deitel.videoplayer requires org.controlsfx.controls file:///Users/
   pauldeitel/Documents/examples/ch21/VideoPlayer/mods/controlsfx.jar
javafx.media requires javafx.base file:///Users/pauldeitel/Downloads/
   javafx-sdk-23.0.1/lib/javafx.base.jar
javafx.media requires javafx.graphics file:///Users/pauldeitel/Downloads/
   javafx-sdk-23.0.1/lib/javafx.graphics.jar
javafx.controls requires javafx.base file:///Users/pauldeitel/Downloads/
   javafx-sdk-23.0.1/lib/javafx.base.jar
javafx.controls requires javafx.graphics file:///Users/pauldeitel/Downloads/
   javafx-sdk-23.0.1/lib/javafx.graphics.jar
...
```

After resolving the root module `com.deitel.videoplayer`, the module resolver looks at each module the root requires, then each module those modules require, and so on, essentially building the module graph shown in Section 21.7.5. If the module resolver finds all the required modules, the program executes. Otherwise, the module resolver throws a `java.lang.module.FindException` and the program terminates.

Generative AI

1 Prompt genAIs to discuss the benefits of using custom Java runtimes in terms of performance, security and reduced size.

21.9 Services and `ServiceLoader`

Section 9.13 discussed "programming to an interface, not an implementation" to create loosely coupled objects. In this section, we discuss services and the `ServiceLoader` class, which help you create loosely coupled system components, making large-scale systems easier to develop and maintain.

`MathTutor` App

We'll create a `MathTutor` app consisting of three modules. The app supports various types of randomly generated math problems. Rather than hard-coding these into the app, we'll load math problems through a service-provider interface that describes how to obtain math problems. We'll then define two service-provider implementation classes that implement this interface—one for addition problems and one for multiplication problems. We'll load and use objects of these service-provider implementation classes at runtime. The following diagram contains the completed three-module app structure:

`MathTutor` App's Modules

Module `com.deitel.mathtutor` aggregates two related packages:

- `com.deitel.mathtutor`: This package contains class `MathTutor`—a command-line app that displays random math problems to the user, inputs the user's responses and displays whether each response is correct or incorrect.
- `com.deitel.mathtutor.spi`: This package contains the service-provider interface `ProblemProvider` and its supporting `abstract` class `Problem`, representing a math problem. `MathTutor` uses `ProblemProvider`s to obtain `Problem` objects.

Module `com.deitel.additionprovider` contains a package of the same name in which we declare class `AdditionProblemProvider`. This implementation of the service-provider interface `ProblemProvider` generates random addition `Problem`s.

Module `com.deitel.multiplicationprovider` contains a package of the same name in which we declare class

`MultiplicationProblemProvider`. This implementation of the service-provider interface `ProblemProvider` generates random multiplication `Problem`s.

How We'll Demonstrate the App

We'll run the `MathTutor` app several times:

- First, we'll run it without placing the service-provider implementation modules on the module path to demonstrate what happens when no service providers are found at runtime.
- Next, we'll "plug in" the `com.deitel.additionprovider` module on the module path, then re-run the app to demonstrate that we can obtain `Problem`s from an `AdditionProblemProvider`.
- Finally, we'll "plug in" both modules `com.deitel.additionprovider` and `com.deitel.multiplicationprovider` on the module path, then re-run the app to demonstrate that we're able to obtain addition and multiplication `Problem`s from the `AdditionProblemProvider` and the `MultiplicationProblemProvider`.

Plug-in Architecture

The `MathTutor` app is easy to extend via this "plug-in" architecture. Simply create a module containing a `ProblemProvider` implementation, then add it to the module path when you run the app. This architecture also makes the app more configurable—you can choose which modules to include on the module path when you execute the app.

Reliable Configuration

The mechanisms for creating loosely coupled systems like the `MathTutor` app have been used extensively in Java since its early versions. A key concept in JPMS—which also applies to modules in general—is reliable configuration. For the `MathTutor` app to be able to display `Problem`s to the user, it must be able to locate and load `ProblemProvider` implementations. As you'll learn, module declarations enable you to specify the service-provider interfaces a module uses and whether a module contains types that implement those interfaces.

21.9.1 Service-Provider Interface

The package `com.deitel.mathtutor.spi` contains the `com.deitel.mathtutor` module's service-provider interface `ProblemProvider` and the supporting `abstract` class `Problem`. The final component of this package's name—`spi`—is commonly used in packages that declare one or more service-provider interfaces. The `ProblemProvider` interface (Fig. 21.9) declares the abstract `getProblem` method (line 6), which returns a `Problem`.

Click here to view code image

```
1  // Fig. 21.9: ProblemProvider.java
2  // Service-provider interface for obtaining a Problem
3  package com.deitel.mathtutor.spi;
4
5  public interface ProblemProvider {
6     public Problem getProblem();
7  }
```

Fig. 21.9 | Service-provider interface for obtaining a `Problem`.

The `abstract Problem` class (Fig. 21.10) provides the common features of this exam-ple's math problems. Each has two `int` operands and an `int` result as well as a `String` representing the operation—the `MathTutor` displays this `String` with each math problem it presents to the user. Class `Problem`'s `abstract` method `getResult` is overridden in each service-provider implementation's concrete `Problem` subclass.

Click here to view code image

```
1  // Fig. 21.10: Problem.java
2  // Problem superclass that contains information about a math problem.
3  package com.deitel.mathtutor.spi;
4
5  public abstract class Problem {
6     private int leftOperand;
7     private int rightOperand;
8     private int result;
9     private String operation;
10
11    // constructor
12    public Problem(int leftOperand, int rightOperand, String operation) {
13       this.leftOperand = leftOperand;
14       this.rightOperand = rightOperand;
15       this.operation = operation;
16    }
```

```
17
18      // gets the leftOperand
19      public int getLeftOperand() {return leftOperand;}
20
21      // gets the rightOperand
22      public int getRightOperand() {return rightOperand;}
23
24      // gets the operation
25      public String getOperation() {return operation;}
26
27      // gets the result
28      public abstract int getResult();
29   }
```

Fig. 21.10 | `Problem` superclass that contains information about a math problem.

21.9.2 Loading and Consuming Service Providers

Class `MathTutor` (Fig. 21.11) is the app's entry point. It provides the logic to locate and load `ProblemProvider` implementations then use them to generate math problems. We split the `MathTutor` class into parts for discussion purposes.

Click here to view code image

```
1   // Fig. 21.11: MathTutor.java
2   // Math tutoring app using ProblemProviders to display math problems.
3   package com.deitel.mathtutor;
4
5   import java.util.List;
6   import java.util.Scanner;
7   import java.util.ServiceLoader;
8   import java.util.ServiceLoader.Provider;
9   import java.util.random.RandomGenerator;
10  import java.util.stream.Collectors;
11  import com.deitel.mathtutor.spi.Problem;
12  import com.deitel.mathtutor.spi.ProblemProvider;
13
14  public class MathTutor {
15     private static Scanner input = new Scanner(System.in);
16
17     public static void main(String[] args) {
```

Fig. 21.11 | Math tutoring app using `ProblemProvider`s to display

math problems.

Using `ServiceLoader` to Locate Service Providers

Lines 19–20 create a **`ServiceLoader`** (package `java.util`) that loads `ProblemProvider` implementations.

Click here to view code image

```
18          // get a service loader for ProblemProviders
19          ServiceLoader<ProblemProvider> serviceLoader =
20             ServiceLoader.load(ProblemProvider.class);
21
```

`ServiceLoader`'s `static` `load` method receives as its argument the `Class` object representing the service-provider interface's type—the compiler automatically creates an object of type `Class<ProblemProvider>` from the class literal `ProblemProvider.class`. Method `load` returns a `ServiceLoader<ProblemProvider>` that dynamically discovers and loads `ProblemProvider` implementations available on the module path.

There are several ways to get service-provider implementations from a `Service-Loader`. Lines 23–24 obtain a `List` of the available service-provider implementations using **`ServiceLoader`'s `stream` method**.

Click here to view code image

```
22          // get the list of service providers
23          List<Provider<ProblemProvider>> providersList =
24             serviceLoader.stream().collect(Collectors.toList());
25
26          // check whether there are any providers
27          if (providersList.isEmpty()) {
28             System.out.println(
29                "Terminating MathTutor: No problem providers found.");
30             return;
31          }
32
```

This returns a `Stream<Provider<ProblemProvider>>` representing all available `Problem-Provider` implementations, if any. Interface `Provider` (imported at line 8) is a nested type of class `ServiceLoader`. For each available `ProblemProvider` implementation, the stream contains one

`Provider<ProblemProvider>` object. Line 24 uses `Stream`'s `collect` method and the predefined `Collector` defined by `Collectors.toList` to get the `List` containing all the available implementations. If that `List` is empty (line 27) the program displays an appropriate message and terminates.

Using a Service-Provider Interface

If the `List` contains any service-provider implementations, lines 36–46 use them to display one math problem at a time to the user. Lines 38–39 randomly select one object from the `providersList`, then invoke that object's `get` method to obtain its `ProblemProvider`. Line 42 then gets a `Problem` from that `ProblemProvider`. Note the loose coupling. The `MathTutor` app does not explicitly mention the `ProblemProvider` implementation classes `AdditionProblemProvider` and `MultiplicationProblemProvider`.

Click here to view code image

```
33          boolean shouldContinue = true;
34          var random = RandomGenerator.getDefault();
35
36          do {
37             // choose a ProblemProvider at random
38             ProblemProvider provider =
39                providersList.get(random.nextInt(providersList.size())).get();
40
41             // get the Problem
42             Problem problem = provider.getProblem();
43
44             // display the problem to the user
45             showProblem(problem);
46          } while (playAgain());
47       }
48
49       // show the math problem to the user
50       private static void showProblem(Problem problem) {
51          var problemStatement = String.format("%nWhat is %d %s %d? ",
52             problem.getLeftOperand(), problem.getOperation(),
53             problem.getRightOperand());
54
55          // display problem and get answer from user
56          System.out.printf(problemStatement);
57          int answer = input.nextInt();
58
59          while (answer != problem.getResult()) {
60             System.out.println("Incorrect. Please try again.");
61             System.out.printf(problemStatement);
```

```
62             answer = input.nextInt();
63          }
64
65          System.out.println("Correct!");
66       }
67
68       // play again?
69       private static boolean playAgain() {
70          System.out.printf("%nTry another? y to continue, n to terminate: ");
71          String response = input.next();
72
73          return response.toLowerCase().startsWith("y");
74       }
75    }
```

Generative AI

1 Prompt genAIs to list the different ways to get service-provider implementations from a JPMS `ServiceLoader`.

21.9.3 `uses` Module Directive and Service Consumers

Figure 21.12 contains the `com.deitel.mathtutor` module declaration. This module `exports` the `com.deitel.mathtutor.spi` package containing the service-provider interface `ProblemProvider` and its supporting `Problem` class, enabling modules that implement `ProblemProvider` to access those types. The new feature in this declaration is the `uses` module directive (line 6), indicating that code in the `com.deitel.mathtutor` module uses `ProblemProvider` objects. Such a module is a **service consumer**.

Click here to view code image

```
1  // Fig. 21.12: module-info.java
2  // Module declaration for the com.deitel.mathtutor module
3  module com.deitel.mathtutor {
4     exports com.deitel.mathtutor.spi; // package for provider interface
5
6     uses com.deitel.mathtutor.spi.ProblemProvider;
7  }
```

Fig. 21.12 | Module declaration for the `com.deitel.mathtutor` module.

For the `MathTutor` to consume `ProblemProvider`s, the `ServiceLoader` must be able to locate and load their implementations dynamically using Java's reflection capabilities. When you run this app, the module resolver will note from the module descriptor (the compiled version of `module-info.java`) that this module uses `ProblemProvider` implementations and thus is dependent on such providers. The module resolver will search the modules on the module path looking for any that provide `ProblemProvider` implementations. If it finds any, it will add them to the module-dependency graph.

21.9.4 Running the App with No Service Providers

To compile the `com.deitel.mathtutor` module, enter:

Click here to view code image

```
javac -d mods/com.deitel.mathtutor \
   src/com.deitel.mathtutor/module-info.java \
   src/com.deitel.mathtutor/com/deitel/mathtutor/MathTutor.java \
   src/com.deitel.mathtutor/com/deitel/mathtutor/spi/*.java
```

Next, run the app with no `ProblemProvider` implementations on the module path by using the following `java` command, which places only the `com.deitel.mathtutor` module on the module path:

Click here to view code image

```
java --module-path mods/com.deitel.mathtutor \
   -m com.deitel.mathtutor/com.deitel.mathtutor.MathTutor
```

The application outputs:

Click here to view code image

```
Terminating MathTutor: No problem providers found.
```

21.9.5 Implementing a Service Provider

Next, let's create class `AdditionProblemProvider` (Fig. 21.13), which implements the `ProblemProvider` service-provider interface (line 10). Package `com.deitel.addition-provider` will be placed in module `com.deitel.additionprovider` (Section 21.9.6). We `import`

interface `ProblemProvider` and class `Problem` from the `com.deitel.mathtutor` module's exported `com.deitel.mathtutor.spi` package (lines 7–8). When `MathTutor` calls an `AdditionProblemProvider`'s `getProblem` method (lines 14–23), the method creates an object of an anonymous inner class (Section 11.5.2) that's extends `Problem` (lines 16–22), passing to `Problem`'s constructor two random `int` operands and the `String` `"+"` to represent the arithmetic operation. Lines 18–21 override `Problem`'s `getResult` method, which adds the left and right operands and returns their sum.

Click here to view code image

```
1  // Fig. 21.13: AdditionProblemProvider.java
2  // AdditionProblemProvider implementation of interface
3  // ProblemProvider for the MathTutor app.
4  package com.deitel.additionprovider;
5
6  import java.util.random.RandomGenerator;
7  import com.deitel.mathtutor.spi.Problem;
8  import com.deitel.mathtutor.spi.ProblemProvider;
9
10 public class AdditionProblemProvider implements ProblemProvider {
11    private static RandomGenerator random = RandomGenerator.getDefault();
12
13    // returns a new addition problem
14    @Override
15    public Problem getProblem() {
16       return new Problem(random.nextInt(10), random.nextInt(10), "+") {
17          // override getResult to add the operands
18          @Override
19          public int getResult() {
20             return getLeftOperand() + getRightOperand();
21          }
22       };
23    }
24 }
```

Fig. 21.13 | `AdditionProblemProvider` implementation of interface `ProblemProvider` for the `MathTutor` app.

21.9.6 `provides...with` Module Directive and Declaring a Service Provider

Figure 21.14 contains the `com.deitel.additionprovider` module declaration. Note that this module `requires` the module

`com.deitel.mathtutor`. Recall from Fig. 21.12 that this module `exports` the package `com.deitel.mathtutor.spi` containing the types used in class `AdditionProblemProvider`.

Click here to view code image

```
1  // Fig. 21.14: module-info.java
2  // Module declaration for the com.deitel.additionprovider module
3  module com.deitel.additionprovider {
4     requires com.deitel.mathtutor;
5
6     provides com.deitel.mathtutor.spi.ProblemProvider
7        with com.deitel.additionprovider.AdditionProblemProvider;
8  }
```

Fig. 21.14 | Module declaration for the `com.deitel.additionprovider` module.

The new feature in this module declaration is the `provides...with` module directive, indicating that the `com.deitel.additionprovider` module is a **service provider**. The `provides` keyword (line 6) is followed by the fully qualified name of an interface or `abstract` class that another module specified in a `uses` module directive. The `with` keyword (line 7) is followed by the name of a concrete class that `implements` the interface or `extends` the `abstract` class. Lines 6–7 specify that this module

- `provides` an implementation of the `ProblemProvider` interface declared in the `com.deitel.mathtutor` module's `com.deitel.mathtutor.spi` package
- `with` the `AdditionProblemProvider` class, which is declared in this module's `com.deitel.additionprovider` package.

21.9.7 Running the App with One Service Provider

Next, we'll run the app with the `AdditionProblemProvider` included in the module path. First, compile the `com.deitel.additionprovider` module, as follows:[29]

29. There can be no spaces after the last line-continuation character and the file name `AdditionProblemProvider.java`, as the filename is part of a path that does not contain spaces.

Click here to view code image

```
javac --module-path mods -d mods/com.deitel.additionprovider \
   src/com.deitel.additionprovider/module-info.java \
   src/com.deitel.additionprovider/com/deitel/additionprovider/\
AdditionProblemProvider.java
```

Then run the app with the following `java` command:

Click here to view code image

```
java --module-path mods \
   -m com.deitel.mathtutor/com.deitel.mathtutor.MathTutor
```

The following is a sample output containing addition problems:

Click here to view code image

```
What is 9 + 6? 15
Correct!

Try another? y to continue, n to terminate: y

What is 2 + 6? 7
Incorrect. Please try again.

What is 2 + 6? 8
Correct!

Try another? y to continue, n to terminate: n
```

21.9.8 Implementing a Second Service Provider

Class `MultiplicationProblemProvider` (Fig. 21.15) also implements the service-provider interface `ProblemProvider` (line 10). `MultiplicationProblemProvider` is nearly identical to `AdditionProblemProvider`, but line 16 passes the `String "*"` for the `Problem`'s operation, and the overridden `Problem` method `getResult` multiplies the left and right operands, returning their product.

Click here to view code image

```
1  // Fig. 21.15: MultiplicationProblemProvider.java
2  // MultiplicationProblemProvider implementation of interface
3  // ProblemProvider for the MathTutor app.
4  package com.deitel.multiplicationprovider;
5
```

```
6  import java.util.random.RandomGenerator;
7  import com.deitel.mathtutor.spi.Problem;
8  import com.deitel.mathtutor.spi.ProblemProvider;
9
10 public class MultiplicationProblemProvider implements ProblemProvider {
11    private static RandomGenerator random = RandomGenerator.getDefault();
12
13    // returns a new multiplication problem
14    @Override
15    public Problem getProblem() {
16       return new Problem(random.nextInt(10), random.nextInt(10), "*") {
17          // override getResult to add the operands
18          @Override
19          public int getResult() {
20             return getLeftOperand() * getRightOperand();
21          }
22       };
23    }
24 }
```

Fig. 21.15 | `MultiplicationProblemProvider` implementation of interface `ProblemProvider` for the `MathTutor` app.

Figure 21.16 contains the `com.deitel.multiplicationprovider` module declaration. Again, this module `requires` the module `com.deitel.mathtutor`. Lines 6–7 specify that this module `provides` an implementation of the `ProblemProvider` interface `with` class `MultiplicationProblemProvider`.

Click here to view code image

```
1  // Fig. 21.16: module-info.java
2  // Module declaration for the com.deitel.multiplicationprovider module
3  module com.deitel.multiplicationprovider {
4     requires com.deitel.mathtutor;
5
6     provides com.deitel.mathtutor.spi.ProblemProvider with
7        com.deitel.multiplicationprovider.MultiplicationProblemProvider;
8  }
```

Fig. 21.16 | Module declaration for the `com.deitel.multiplicationprovider` module.

21.9.9 Running the App with Two Service Providers

Finally, let's run the app with the `AdditionProblemProvider` and

the `Multiplication-ProblemProvider` service-provider implementations included in the module path. First, compile the `com.deitel.multiplicationprovider` module, as follows:

Click here to view code image

```
javac --module-path mods \
   -d mods/com.deitel.multiplicationprovider \
   src/com.deitel.multiplicationprovider/module-info.java \
   src/com.deitel.multiplicationprovider/com/deitel/\
multiplicationprovider/MultiplicationProblemProvider.java
```

Run the app with the following `java` command:

Click here to view code image

```
java --module-path mods \
   -m com.deitel.mathtutor/com.deitel.mathtutor.MathTutor
```

The following is a sample output containing both addition and multiplication problems:

Click here to view code image

```
What is 4 * 8? 20
Incorrect. Please try again.

What is 4 * 8? 32
Correct!

Try another? y to continue, n to terminate: y

What is 3 * 6? 18
Correct!

Try another? y to continue, n to terminate: y

What is 3 + 7? 10
Correct!

Try another? y to continue, n to terminate: y

What is 9 + 3? 12
Correct!

Try another? y to continue, n to terminate: n
```

Generative AI

1 Prompt genAIs to list the benefits and disadvantages of JPMS's service-provider architecture.

2 Prompt genAIs for real-world use-cases of JPMS services and service providers.

21.10 Wrap-Up

This chapter presented the Java Platform Module System (JPMS) and discussed its role in creating more maintainable and secure Java applications. You learned that modules are uniquely named, reusable groups of related packages and resources. You used module declarations in `module-info.java` files to specify a module's name, dependencies, exported packages, services and reflection permissions.

You learned that all modules implicitly depend on `java.base`, which contains fundamental packages required by all Java application. You created module declarations using:

- `requires` directives for specifying dependencies,
- `exports` and `exports...to` directives for making packages available to other modules,
- the `open` modifier and the `opens` and `opens...to` directives for controlling reflection access,
- `uses` for consuming services and
- `provides...with` for implementing services.

You explored how module declarations enforce strong encapsulation, ensuring that only explicitly exported packages are accessible to other modules.

We built modular applications and used the Graphviz tool to visualize their dependencies in module-dependency graphs. You used the `jdeps` tool to analyze module dependencies and learned how the module resolver ensures dependencies are fulfilled at runtime. You used `jlink` to create custom runtime images containing only the modules an application needs, making Java more suitable for resource-constrained and cloud environments.

We discussed migrating existing code to modules using unnamed modules and automatic modules and how to use jdeps to check whether legacy code uses internal APIs that are now strongly encapsulated. We presented services, service providers and using

the class `ServiceLoader` to create loosely coupled system components that make large-scale applications easier to develop and maintain. We then presented a `MathTutor` application in which we implemented service-provider interfaces and created multiple service providers, demonstrating how JPMS enables plug-in architectures and extensible applications. This foundation in JPMS prepared you to build modern Java applications that are more secure, maintainable, and efficient. In the next chapter, we discuss miscellaneous additional Java topics.

Special Section: Generative AI Exercises

For each of the following questions, try multiple genAIs to determine which performs best. Always verify the results. Keep in mind that they sometimes make mistakes and even hallucinate, creating unreliable information.

21.1 ***(JPMS Benefits and Trade-Offs)*** Prompt genAIs, asking for the benefits and tradeoffs of modularizing legacy Java applications.

21.2 ***(JPMS Best Practices)*** Prompt genAIs to provide a list of JPMS best practices for modularizing Java applications.

21.3 ***(JPMS Pitfalls)*** Prompt genAIs for a list of common JPMS pitfalls when modularizing Java applications.

21.4 ***(JPMS Troubleshooting Tips)*** Prompt genAIs for a list of JPMS troubleshooting tips.

21.5 ***(JPMS Security Best Practices)*** Prompt genAIs to provide a list of JPMS security best practices when opening modules to other modules for reflection.

21.6 ***(Granting Temporary Access to a Module)*** Software-testing frameworks, like JUnit, are commonly used in real-world Java development. Prompt genAIs, asking how to provide software-testing frameworks with temporary access to your Java modules' contents.

21.7 ***(`java.base` Module)*** Prompt genAIs, asking why the `java.base` module contains so many packages.

21.8 ***(Multi-release JARs)*** Prompt genAIs for an introduction to multi-release JAR files in Java and how they're used with JPMS.

21.9 ***(`jpackage` Tool)*** Prompt genAIs for a tutorial on using the JDK's `jpackage` tool to create self-contained Java applications that

can be installed on various platforms.

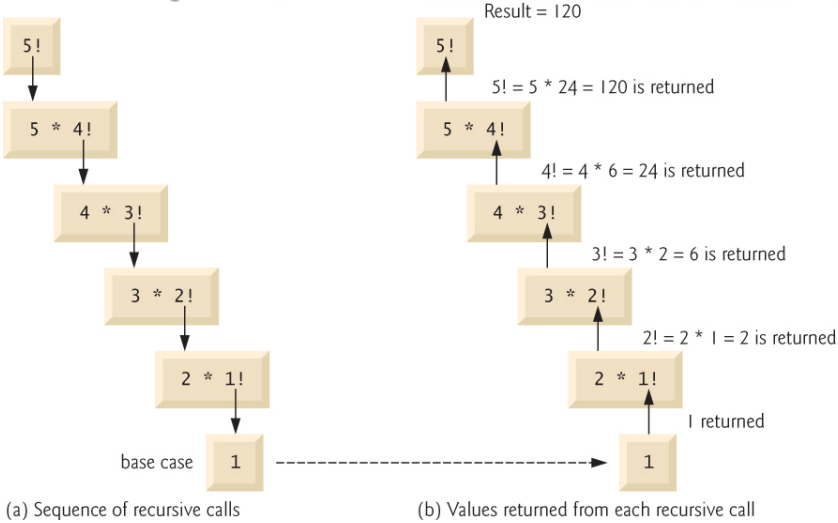
21.10 ***(Modular Software Design)*** Prompt genAIs, asking for real-world examples of modular software design using JPMS.

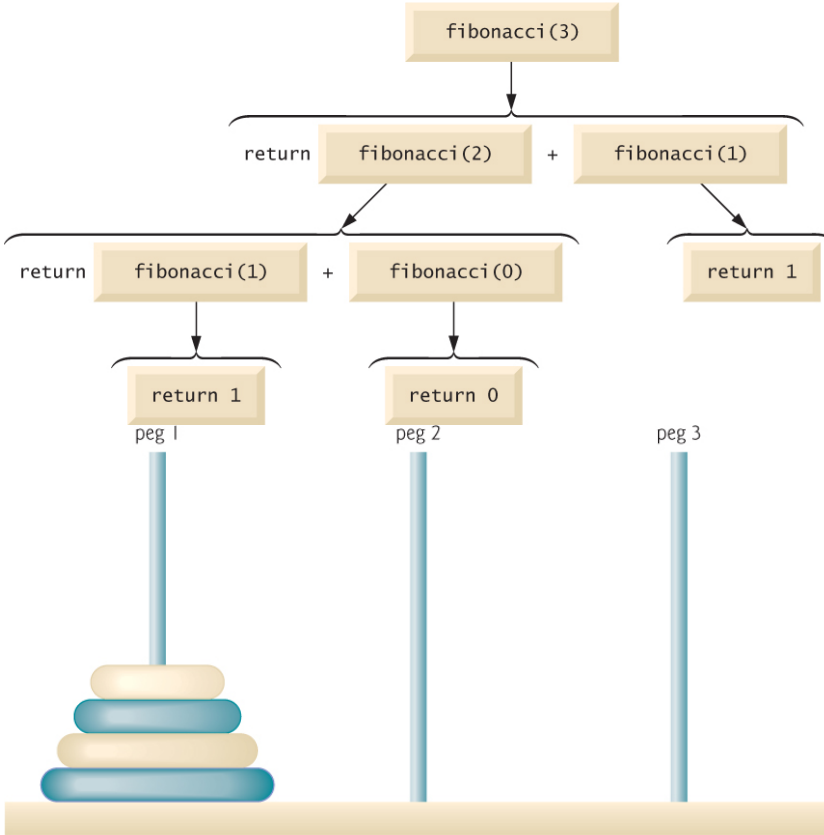
21.11 ***(Step-By-Step Modularization Migration Guide)*** Prompt genAIs with an existing Java application and ask them to provide a step-by-step migration guide for modularizing the app using JPMS.

21.12 ***(JPMS and Java Security)*** Prompt genAIs for a detailed discussion of how JPMS improves Java security.

21.13 ***(Real-World Use-Cases of `ServiceLoader`)*** Prompt genAIs for real-world use cases for `ServiceLoader` and how it supports loose coupling in modular applications.

Result = 120


5!
5 * 4!
4 * 3!
3 * 2!
2 * 1!
base case
1
(a) Sequence of recursive calls
5!
5! = 5 * 24 = 120 is returned
5 * 4!
4! = 4 * 6 = 24 is returned
4 * 3!
3! = 3 * 2 = 6 is returned
3 * 2!
2! = 2 * 1 = 2 is returned
2 * 1!
1 returned
1
(b) Values returned from each recursive call



(a) Level 0

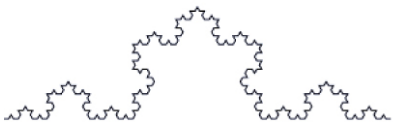
(b) Level 1

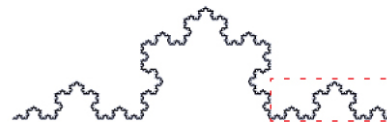
(c) Level 2

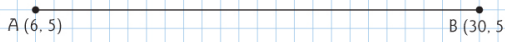
(d) Level 3

(e) Level 4

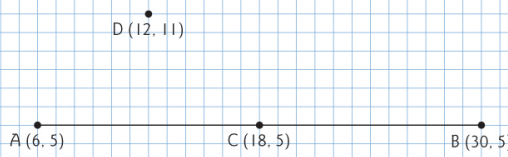
(f) Level 5



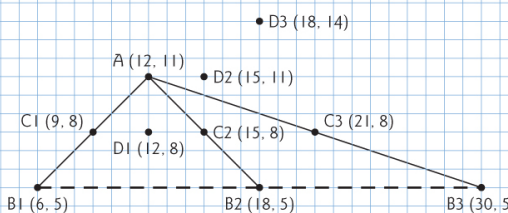




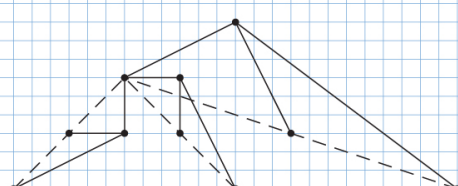
Origin (0, 0)



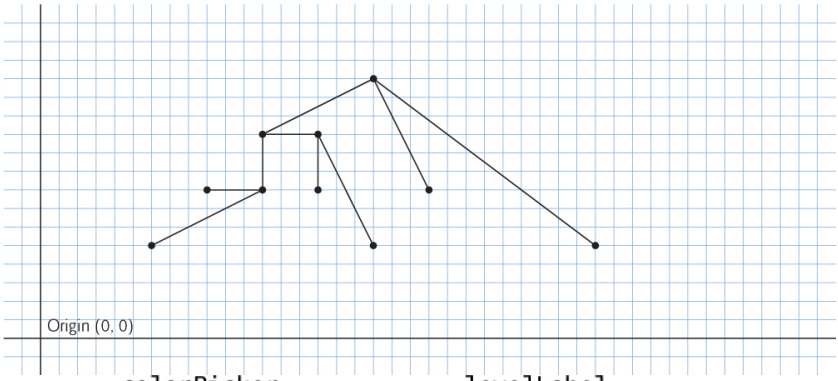
Origin (0, 0)

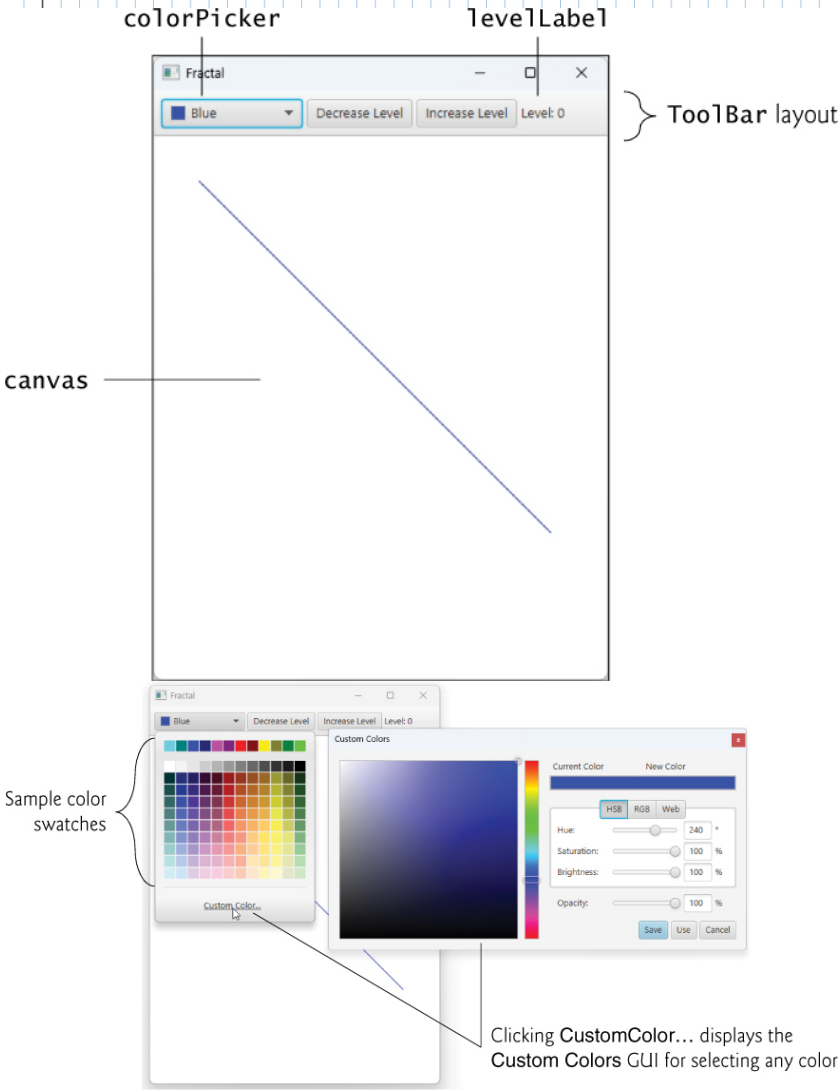


Origin (0, 0)



Origin (0, 0)


Origin (0, 0)


colorPicker
levelLabel
Fractal
Blue
Decrease Level
Increase Level
Level: 0
ToolBar layout
canvas
Sample color swatches
Custom Color...
Custom Colors
Current Color
New Color
HSB
RGB
Web
Hue:
Saturation:
Brightness:
Opacity:
Save
Use
Cancel
Clicking CustomColor... displays the Custom Colors GUI for selecting any color

Recursion and Big *O*

Objectives

In this chapter, you'll:

- Write and use recursive methods.
- Determine the base case and recursion step in a recursive algorithm.
- Learn how the system handles recursive method calls.
- Learn the differences between recursion and iteration and when to use each.
- Learn what fractals are and how to draw them using recursion and JavaFX's `Canvas` and `GraphicsContext` classes.
- Learn what recursive backtracking is and why it's an effective problem-solving technique.
- Use Big *O* notation to compare the efficiency of algorithms.

Outline

22.1 Introduction

22.2 Recursion Concepts

22.3 Recursion Example: Factorials

22.4 Recursion Example: Fibonacci Series

22.5 Recursion vs. Iteration

22.6 Towers of Hanoi

22.7 Fractals

22.7.1 Koch Curve Fractal
22.7.2 (Optional) Case Study: Lo Feather Fractal
22.7.3 (Optional) Fractal App GUI
22.7.4 (Optional) `FractalController` Class
22.8 Recursive Backtracking
22.9 Big O Notation
22.9.1 $O(1)$ Algorithms
22.9.2 $O(n)$ Algorithms
22.9.3 $O(n^2)$ Algorithms
22.10 Common Big O Notations
22.11 Wrap-Up

22.1 Introduction

In this chapter, we concentrate on some key aspects of computer science thinking that go beyond programming fundamentals, including a focus on performance issues. We begin with a treatment of **recursive methods** that call themselves. We'll show examples and compare recursive programming to iterative programming. We'll indicate where each might be preferable. We also introduce **Big *O* notation**, which concisely classifies algorithms by how hard they have to work to get the job done. Big *O* helps you compare the efficiency of algorithms.

22.2 Recursion Concepts

Recursive problem-solving approaches have several elements in common. When a recursive method is called to solve a problem, it can solve only the simplest case(s) or **base case(s)**. If the method is called with a base case, it returns a result. If the method is called with a more complex problem, it divides the problem into two conceptual pieces—a piece that the method knows how to do and a piece that it does not know how to do. To make recursion feasible, the latter piece must resemble the original problem but be a slightly simpler or smaller version. Because this new problem resembles the original problem, the method calls a fresh copy of itself to work on the smaller problem—this is referred to as a **recursive call** and is also called the **recursion step**. The recursion step typically includes

a `return` statement because its result will be combined with the portion of the problem the method knew how to solve to form a result that will be passed back to the original caller. Separating the problem into two smaller portions is a form of the divide-and-conquer approach.

The recursion step executes while the original method call is still active (i.e., it has not finished executing). It can result in many more recursive calls as the method divides each new subproblem into two conceptual pieces. For the recursion to eventually terminate, each time the method calls itself with a simpler version of the original problem, the sequence of smaller and smaller problems must converge on a base case. When the method recognizes the base case, it returns a result to the previous copy of the method. A sequence of returns ensues until the original method call returns the final result to the caller. We'll illustrate this process with a concrete example in Section 22.3.

22.3 Recursion Example: Factorials

Let's write a recursive program to perform a popular mathematical calculation. Consider the factorial of a positive integer n, written $n!$ (pronounced "n factorial"), which is the product

$$n \cdot (n - 1) \cdot (n - 2) \cdot \ldots \cdot 1$$

with 1! equal to 1 and 0! defined to be 1. For example, 5! is the product $5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$, which is equal to 120.

The factorial of an integer `number` (where `number` $\geq$ 0) can be calculated iteratively (nonrecursively) using a `for` statement as follows:

Click here to view code image

```
factorial = 1;

for (int counter = number; counter >= 1; --counter) {
   factorial *= counter;
}
```

A recursive declaration of the factorial calculation for integers greater than 1 is arrived at by observing the following relationship:

$$n! = n \cdot (n - 1)!$$

For example, 5! is clearly equal to 5 · 4!, as shown by the following equations:

$$5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$$
$$5! = 5 \cdot (4 \cdot 3 \cdot 2 \cdot 1)$$
$$5! = 5 \cdot (4!)$$

The evaluation of 5! would proceed as shown in the following diagram. Part (a) contains the succession of recursive calls that proceed until 1! (the base case) is evaluated as 1, which terminates the recursion. Part (b) contains the values returned from each recursive call to its caller until the final value is calculated and returned.

Implementing a Recursive Factorial Function

Figure 22.1 recursively calculates the factorials of 0–10 then 20, 30 and 40. The recursive method `factorial` (lines 7–15) first determines whether a terminating condition (line 8) is `true`. If `so`, `factorial` returns `BigInteger.ONE`, no further recursion is necessary, and the method returns. (A precondition of calling method `factorial` in this example is that its argument must be nonnegative.) If `number` is greater than `1`, lines 12–13 express the problem as the product of `BigInteger.valueOf(number)` and a recursive call to method `factorial` evaluating the factorial of `number - 1`, which is a slightly smaller problem than the original calculation, `factorial(number)`. The loop in lines 19–21 calls the `factorial` function for the values from `0` through `10`. Lines 24–25 call method `factorial` for the values 20, 30 and 40. Note that factorial values grow quickly.

Click here to view code image

```
1  // Fig. 22.1: FactorialCalculator.java
2  // Recursive factorial method.
3  import java.math.BigInteger;
4
5  public class FactorialCalculator {
6     // recursive method factorial (assumes its parameter is >= 0)
7     public static BigInteger factorial(int number) {
8        if (number <= 1) { // test for base case
9           return BigInteger.ONE; // base cases: 0! = 1 and 1! = 1
10       }
```

```
11          else { // recursion step
12             return BigInteger.valueOf(number).multiply(
13                factorial(number - 1));
14          }
15       }
16
17       public static void main(String[] args) {
18          // calculate the factorials of 0 through 10
19          for (int counter = 0; counter <= 10; ++counter) {
20             System.out.printf("%d! = %d%n", counter, factorial(counter));
21          }
22
23          // display the factorials of 20, 30 and 40
24          System.out.printf("%n20!: %s%n30!: %s%n40!: %s%n",
25             factorial(20), factorial(30), factorial(40));
26       }
27    }
```

```
0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800
```

```
20!: 2432902008176640000
30!: 265252859812191058636308480000000
40!: 815915283247897734345611269596115894272000000000
```

Fig. 22.1 | Recursive factorial method.

Factorial Values Grow Quickly

Method `factorial` receives an `int` and returns a `BigInteger` object (introduced in Section 3.12). The method produces large values so quickly that type `long` does not help us compute many factorial values before reaching its maximum value. Type `long` can store values from –9,223,372,036,854,775,808 to 9,223,372,036,854,775,807—a maximum of 19 decimal digits (which can represent integers in the quintillions). The factorial of 20 is a 19-digit number, and the subsequent values have

significantly more than 19 digits.

Indirect Recursion

A recursive function may call another function, which may, in turn, make a call back to the recursive function. This is known as an **indirect recursive call**. For example, function `A` calls function `B`, which makes a call back to function `A`. This is still recursion because the second call to function `A` is made while the first call to function `A` is still active—that is, the first call to function `A` has not yet finished executing and has not returned to function `A`'s original caller because it's waiting for function `B` to return a result.

Stack Overflow and Infinite Recursion

A computer's memory is finite, so only a certain amount of memory can store activation records on the function-call stack. If more recursive function calls occur than can have their activation records stored on the stack, a fatal error known as stack overflow occurs.[1] This typically results from infinite recursion, which can be caused by omitting the test for the base case or incorrectly writing the recursion step so it does not converge on the base case. Infinite recursion is analogous to an infinite loop in an iterative solution.

1. This is how the website `stackoverflow.com` got its name. This is an excellent site for getting answers to your programming questions.

Generative AI

1 `BigInteger` is slower than `long` for calculations that are within `long`'s range. Prompt genAIs with Fig. 22.1's code, asking them to modify it to perform the same recursive factorial calculations with `BigInteger`s and `long`s and display the calculation-time performance increase as a percentage. Run the generated code and examine the results.

22.4 Recursion Example: Fibonacci Series

The **Fibonacci series**,

Click here to view code image

0, 1, 1, 2, 3, 5, 8, 13, 21, …

begins with 0 and 1 and has the property that each subsequent Fibonacci number is the sum of the previous two. This series occurs in nature and describes a form of spiral. The ratio of successive Fibonacci numbers converges on a constant value of 1.618…, a number that has been called the **golden ratio** or the **golden mean.** Humans tend to find the golden mean aesthetically pleasing. Architects often design windows, rooms and buildings whose length and width are in the ratio of the golden mean. Postcards are often designed with a golden-mean length-to-width ratio.

Recursive Fibonacci Definition

The Fibonacci series may be defined recursively as follows:

Click here to view code image

$$\text{fibonacci}(0) = 0$$
$$\text{fibonacci}(1) = 1$$
$$\text{fibonacci}(n) = \text{fibonacci}(n-1) + \text{fibonacci}(n-2)$$

with two base cases—`fibonacci(0)` is defined as 0, and `fibonacci(1)` is defined as 1.

Implementing the Recursive Fibonacci Algorithm

Figure 22.2 recursively calculates the *i*th Fibonacci number, using method `fibonacci` (lines 9–18). Method `main` (lines 20–26) tests `fibonacci`, displaying the Fibonacci values of 0–40. The variable `counter` (line 22) indicates which Fibonacci number to calculate in each loop iteration. Fibonacci numbers become large quickly (though not as quickly as factorials), so we use `BigInteger` as the parameter type and the return type of the `fibonacci` method.

Click here to view code image

```
1  // Fig. 22.2: FibonacciCalculator.java
2  // Recursive fibonacci method.
3  import java.math.BigInteger;
4
5  public class FibonacciCalculator {
6     private static BigInteger TWO = BigInteger.valueOf(2);
7
8     // recursive declaration of method fibonacci
9     public static BigInteger fibonacci(BigInteger number) {
```

```
10          if (number.equals(BigInteger.ZERO) ||
11              number.equals(BigInteger.ONE)) { // base cases
12             return number;
13          }
14          else { // recursion step
15             return fibonacci(number.subtract(BigInteger.ONE)).add(
16                fibonacci(number.subtract(TWO)));
17          }
18       }
19
20       public static void main(String[] args) {
21          // displays the fibonacci values from 0-40
22          for (int counter = 0; counter <= 40; ++counter) {
23             System.out.printf("Fibonacci of %d is: %d%n", counter,
24                fibonacci(BigInteger.valueOf(counter)));
25          }
26       }
27    }
```

```
Fibonacci of 0 is: 0
Fibonacci of 1 is: 1
Fibonacci of 2 is: 1
Fibonacci of 3 is: 2
Fibonacci of 4 is: 3
Fibonacci of 5 is: 5
Fibonacci of 6 is: 8
Fibonacci of 7 is: 13
Fibonacci of 8 is: 21
Fibonacci of 9 is: 34
Fibonacci of 10 is: 55
...
Fibonacci of 37 is: 24157817
Fibonacci of 38 is: 39088169
Fibonacci of 39 is: 63245986
Fibonacci of 40 is: 102334155
```

Fig. 22.2 | Recursive fibonacci method.

The call to method `fibonacci` (line 24) from `main` is not a recursive call, but all subsequent calls to `fibonacci` from lines 15–16 are recursive because the calls are initiated by method `fibonacci`. Each time `fibonacci` is called, it immediately tests for the base cases—`number` equal to 0 or `number` equal to 1 (lines 10–11). We use `BigInteger` constants `ZERO` and `ONE` to represent 0 and 1, respectively. If the condition in lines 10–11 is `true`, `fibonacci` simply returns `number`, because `fibonacci(0)` is 0 and `fibonacci(1)` is 1. Interestingly, if `number` is greater than 1, the recursion step generates *two* recursive calls (lines 15–16), each for a

slightly smaller problem than the original call to `fibonacci`. Lines 15–16 use `BigInteger` methods `add` and `subtract` to help implement the recursive step. We also use a constant of type `BigInteger` named `TWO` (defined at line 6).

Analyzing the Calls to Method `Fibonacci`

The following diagram presents how function `fibonacci` evaluates `fibonacci(3)`:

- At the diagram's bottom, we're left with the values 1, 0 and 1 —the results of evaluating the base cases.
- The first two return values (from left to right), 1 and 0, are returned as the values for the calls `fibonacci(1)` and `fibonacci(0)`.
- The sum 1 plus 0 is returned as `fibonacci(2)`'s value. This is added to the result (1) of the rightmost call to `fibonacci(1)`, producing the value 2.

This final value is then returned as `fibonacci(3)`'s value.

This diagram raises some interesting issues about the order in which Java compilers evaluate the operands of operators. This order differs from that in which operators are applied to their operands—namely, the order dictated by the operator precedence rules. From the diagram, it appears that while `fibonacci(3)` is being evaluated, two recursive calls will be made—`fibonacci(2)` and `fibonacci(1)`. But in what order will they be made? The Java language specifies that the operands' order of evaluation is left to right. Thus, the call `fibonacci(2)` is made first and `fibonacci(1)` second.

Complexity Issues

A word of caution is in order about recursive programs like the one we use here to generate Fibonacci numbers. Each invocation of the `fibonacci` function that does not match one of the base cases (0 or 1) results in two more recursive calls to the `fibonacci` function. Hence, this set of recursive calls rapidly gets out of hand. Calculating the Fibonacci value of 20 with the recursive implementation requires 21,891 calls to the `fibonacci` function; calculating the Fibonacci value of 30 requires 2,692,537 calls!

As you try to calculate larger Fibonacci values, you'll notice that each consecutive Fibonacci number you calculate results in a

substantial increase in calculation time and in the number of calls to the `fibonacci` function. For example, the Fibonacci value of 31 requires 4,356,617 calls, and the Fibonacci value of 32 requires 7,049,155 calls! The number of calls to `fibonacci` increases quickly—1,664,080 more calls between Fibonacci values of 30 and 31 and 2,692,538 more calls between Fibonacci values of 31 and 32! The difference in the number of calls made between Fibonacci values of 31 and 32 is more than 1.5 times the difference in the number of calls for Fibonacci values between 30 and 31. Problems of this nature can humble even the world's most powerful computers.

In the field of complexity theory, computer scientists study how hard algorithms work to complete their tasks—that is, how many operations do they perform? We introduce various complexity issues in Section 22.9.

Avoid Fibonacci-style recursion that results in an exponential "explosion" of method calls taking a lot of time and consuming a lot of memory on the function-call stack. Iterative solutions in these cases are much more efficient.

Generative AI

1 Prompt genAIs, asking for examples of how the Fibonacci series occurs in nature and for real-world applications of the series in computer programming.

2 Prompt genAIs with Fig. 22.2's code, asking them to modify it to perform the same recursive Fibonacci calculations with `BigInteger`s and `long`s and display the calculation-time performance increase as a percentage. Run the generated code and examine the results.

3 Prompt genAIs for an explanation of the performance technique called memoization. Next, ask them to modify Fig. 22.2's code to perform the recursive Fibonacci calculations using the non-memoized and memoized approaches, then display the calculation-time performance increase as a percentage. Run the generated code and examine the results.

22.5 Recursion vs. Iteration

We've studied methodso calculate factorials and Fibonacci values. Each can easily be implemented recursively or iteratively. Let's compare the two approaches and discuss why you might choose each approach in a particular situation.

To illustrate the differences between iteration and recursion, let's examine an iterative factorial calculation (Fig. 22.3, lines 11–13). Here, we use an iteration statement rather than a selection statement (Fig. 22.1, lines 8–14). Both solutions use a termination test. In the recursive solution (Fig. 22.1), line 8 tests for the base case. In Fig. 22.3's iterative solution, line 11 tests the loop-continuation condition—if the test fails, the loop terminates. Finally, instead of producing smaller versions of the original problem, the iterative solution uses a counter-controlled loop.

Click here to view code image

```
1  // Fig. 22.3: FactorialCalculator.java
2  // Iterative factorial method.
3  import java.math.BigInteger;
4
5  public class FactorialCalculator {
6     // iterative declaration of method factorial
7     public static BigInteger factorial(int number) {
8        var result = BigInteger.ONE;
9
10       // iteratively calculate factorial
11       for (int i = number; i >= 1; --i) {
12          result = result.multiply(BigInteger.valueOf(i));
13       }
14
15       return result;
16    }
17
18    public static void main(String[] args) {
19       // calculate the factorials of 0 through 10
20       for (int counter = 0; counter <= 10; ++counter) {
21          System.out.printf("%d! = %s%n", counter, factorial(counter));
22       }
23
24       // display the factorials of 20, 30 and 40
25       System.out.printf("%n20!: %s%n30!: %s%n40!: %s%n",
26          factorial(20), factorial(30), factorial(40));
27    }
28 }
```

```
0! = 1
1! = 1
```

```
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800

20!: 2432902008176640000
30!: 265252859812191058636308480000000
40!: 815915283247897734345611269596115894272000000000
```

Fig. 22.3 | Iterative factorial method.

Comparing Iteration and Recursion

Both iteration and recursion are based on a control statement: Iteration uses an iteration statement (e.g., `for`, `while` or `do...while`), whereas recursion uses a selection statement (e.g., `if`, `if...else` or `switch`):

- Both involve iteration: Iteration explicitly uses an iteration statement, whereas recursion achieves iteration through repeated method calls.
- Both involve a termination test: Iteration terminates when the loop-continuation condition fails, whereas recursion terminates when a base case is reached.
- Iteration with counter-controlled iteration and recursion both gradually approach termination: Iteration keeps modifying a counter until it assumes a value that makes the loop-continuation condition fail, whereas recursion keeps producing smaller versions of the original problem until the base case is reached.
- Both can occur infinitely: An infinite loop occurs with iteration if the loop-continuation test never becomes false, whereas infinite recursion occurs if the recursion step does not reduce the problem in a manner that converges on the base case or if the base case is not tested. This results in stack overflow, and the program terminates abruptly (that is, crashes).

Negatives of Recursion

Recursion has many negatives. It repeatedly invokes the mechanism, and consequently the overhead, of method calls. This iteration can be expensive in terms of both processor time and memory space. Each recursive call causes another copy of the method (actually, only the method's variables, stored in the stack frame) to be created. This set of copies can consume considerable memory space. Since iteration occurs in a method, repeated method calls and their memory overhead are avoided.

When to Choose Recursion vs. Iteration

Any problem that can be solved recursively can be solved iteratively and vice versa. A recursive approach is preferred over an iterative approach when the recursive approach more naturally mirrors the problem and results in a program that's easier to understand and debug. A recursive approach can often be implemented with fewer lines of code. Another reason to choose a recursive approach is that an iterative one might not be apparent. When performance is crucial, you might want to try various iterative and recursive approaches to determine which achieves your goal.

Generative AI

1 Prompt genAIs, asking for suggestions on debugging recursive Java methods.

22.6 Towers of Hanoi

Earlier in this chapter, we studied methods that can be implemented recursively or iteratively. Now, we present a problem whose recursive solution demonstrates the elegance of recursion and whose iterative solution may not be as apparent.

The **Towers of Hanoi** is one of the classic recursion problems. Legend has it that in a temple in the Far East, priests are attempting to move a stack of golden disks from one diamond peg to another: The initial stack has 64 disks threaded onto one peg and arranged from bottom to top by decreasing size. The priests are attempting to move the stack from one peg to another under the constraints that exactly one disk is moved at a time and at no time may a larger disk be placed above a smaller disk. Three pegs are provided, one being used for temporarily holding disks. Supposedly, the world will end when the priests complete their tasks, so we have little incentive to

facilitate their efforts.

Let's assume the priests are attempting to move the disks from peg 1 to peg 3. We wish to develop an algorithm that prints the precise sequence of peg-to-peg disk transfers.

If we try to find an iterative solution, we'll likely find ourselves hopelessly "knotted up" in managing the disks. Instead, attacking this problem recursively yields an intuitive solution. Moving n disks can be viewed in terms of moving only $n - 1$ disks (hence the recursion) as follows:

- Move $n - 1$ disks from peg 1 to peg 2, using peg 3 as a temporary holding area.
- Move the last disk (the largest) from peg 1 to peg 3.
- Move $n - 1$ disks from peg 2 to peg 3, using peg 1 as a temporary holding area.

The process ends when the last task involves moving the $n = 1$ disk (i.e., the base case). This task is accomplished simply by moving the disk without using a temporary holding area.

In Fig. 22.4, method `solveTowers` (lines 5–22) solves the Towers of Hanoi, given the total number of disks (we'll call the method for the case with `3` disks), the starting peg, the ending peg, and the temporary holding peg as parameters.

Click here to view code image

```
1  // Fig. 22.4: TowersOfHanoi.java
2  // Towers of Hanoi solution with a recursive method.
3  public class TowersOfHanoi {
4     // recursively move disks between towers
5     public static void solveTowers(int disks, int sourcePeg,
6        int destinationPeg, int tempPeg) {
7        // base case -- only one disk to move
8        if (disks == 1) {
9           System.out.printf("%n%d --> %d", sourcePeg, destinationPeg);
10          return;
11       }
12
13       // recursion step -- move (disk - 1) disks from sourcePeg
14       // to tempPeg using destinationPeg
15       solveTowers(disks - 1, sourcePeg, tempPeg, destinationPeg);
16
17       // move last disk from sourcePeg to destinationPeg
18       System.out.printf("%n%d --> %d", sourcePeg, destinationPeg);
19
```

```
20          // move (disks - 1) disks from tempPeg to destinationPeg
21          solveTowers(disks - 1, tempPeg, destinationPeg, sourcePeg);
22       }
23
24       public static void main(String[] args) {
25          int startPeg = 1; // value 1 used to indicate startPeg in output
26          int endPeg = 3; // value 3 used to indicate endPeg in output
27          int tempPeg = 2; // value 2 used to indicate tempPeg in output
28          int totalDisks = 3; // number of disks
29
30          // initial nonrecursive call: move all disks.
31          solveTowers(totalDisks, startPeg, endPeg, tempPeg);
32       }
33    }
```

```
1 --> 3
1 --> 2
3 --> 2
1 --> 3
2 --> 1
2 --> 3
1 --> 3
```

Fig. 22.4 | Towers of Hanoi solution with a recursive method.

The base case (lines 8–11) occurs when only one disk needs to be moved from the starting peg to the ending peg. The recursion step (lines 15–21) moves `disks - 1` disks (line 15) from the first peg (`sourcePeg`) to the temporary holding peg (`tempPeg`). When all but one of the disks have been moved to the temporary peg, line 18 moves the largest disk from the start peg to the destination peg. Line 21 finishes the rest of the moves by calling the method `solveTowers` to recursively move `disks - 1` disks from the temporary peg (`tempPeg`) to the destination peg (`destinationPeg`), this time using the first peg (`sourcePeg`) as the temporary peg. Line 31 in `main` calls the recursive `solveTowers` method, which outputs the steps to the command prompt.

Generative AI

1 Prompt genAIs for the lore and legend of the Towers of Hanoi problem.

22.7 Fractals

A **fractal** is a geometric figure generated from a pattern repeated recursively. The figure is modified by recursively applying the pattern to each segment of the original figure. Although such figures had been studied before the 20th century, the mathematician Benoit Mandelbrot in the 1970s introduced the term "fractal," along with the specifics of how a fractal is created and the practical applications of fractals. Mandelbrot's fractal geometry provides mathematical models for many complex forms found in nature, such as mountains, clouds and coastlines. Fractals have many uses in mathematics and science. They can be used to better understand systems or patterns that appear in nature (e.g., ecosystems), in the human body (e.g., in the folds of the brain), or in the universe (e.g., galaxy clusters). Not all fractals resemble objects in nature. Drawing fractals has become a popular art form. Fractals have a **self-similar property**—when subdivided into parts, each resembles a reduced-size copy of the whole. Many fractals yield an exact copy of the original when a portion of the fractal is magnified—such a fractal is said to be **strictly self-similar**.

22.7.1 Koch Curve Fractal

As an example, let's look at the strictly self-similar **Koch Curve** fractal—formed by removing the middle third of each line in the drawing and replacing it with two lines that form a point, such that if the middle third of the original line remained, an equilateral triangle would be formed;
Formulas for creating fractals often involve removing all or part of the previous fractal image. This pattern has already been determined for this fractal. We focus here on using those formulas in a recursive solution.

We start with a straight line in Part (a) and apply the pattern, creating a triangle from the middle third, as in Part (b). We then apply the pattern to each straight line, resulting in Part (c). Each time the pattern is applied, we say that the fractal is at a new **level** or **depth**— sometimes, the term **order** is also used. Fractals can be displayed at many levels—for example, a fractal at level 3 has had three iterations of the pattern applied (Part (d)). After only a few iterations, this fractal begins to look like a portion of a snowflake, as in Parts (e) and (f). Since this is a strictly self-similar fractal, each portion contains an exact copy of the fractal. For example, in Part (f), we've highlighted a portion of the fractal with a dashed box. If the image in this box were increased in size, it would look exactly

like the entire fractal of part (e).
The **Koch Snowflake** fractal is similar to the Koch Curve but begins with a triangle rather than a line. The same pattern is applied to each side of the triangle, resulting in an image resembling an enclosed snowflake.

22.7.2 (Optional) Case Study: Lo Feather Fractal

We now demonstrate using recursion to draw fractals by writing a program to create a strictly self-similar fractal. We call this the "Lo feather fractal," named for Sin Han Lo, a Deitel & Associates, Inc. colleague who created it. The fractal will eventually resemble one-half of a feather. The base case, or fractal level of 0, begins as a line between two points, `A` and `B`, as in the following diagram:
To create the next higher level, we find the midpoint (`C`) of the line. To calculate the location of point `C`, use the following formula:

```
xC = (xA + xB) / 2;
yC = (yA + yB) / 2;
```

The `x` and `y` to the left of each letter refer to the *x*-coordinate and *y*-coordinate of that point, respectively. For example, `xA` refers to the *x*-coordinate of point `A`, while `yC` refers to the *y*-coordinate of point `C`. Our diagrams denote the point by its letter, followed by two numbers representing the *x*- and *y*-coordinates.

To create this fractal, we must find a point `D` that lies to the left of segment `AC` and creates an isosceles right triangle `ADC`. To calculate point `D`'s location, use the formulas:

Click here to view code image

```
xD = xA + (xC - xA) / 2 - (yC - yA) / 2;
yD = yA + (yC - yA) / 2 + (xC - xA) / 2;
```

We now move from level 0 to level 1 as follows: First, add points `C` and `D`, as in the following diagram:
Then, remove the original line and add segments `DA`, `DC` and `DB`. The remaining lines will curve at an angle, causing our fractal to look like a feather.

For the next level, this algorithm is repeated on each of the three lines in level 1. For each line, the formulas above are applied, where the former point `D` is now considered to be point `A`, while

the other end of each line is considered to be point `B`. The following diagram contains the line from level 0 (now a dashed line) and the three added lines from level 1:
We've changed point `D` to be point `A`, and the original points `A`, `C` and `B` to `B1`, `B2` and `B3`, respectively. The preceding formulas have been used to find the new points `C` and `D` on each line. These points are also numbered 1–3 to track which point is associated with each line. For example, points `C1` and `D1` represent points `C` and `D` associated with the line formed from points `A` to `B1`.

To achieve level 2, the three lines in the preceding diagram are removed and replaced with new lines from the `C` and `D` points just added. The following diagram contains the new lines (the lines from level 2 are dashed lines for your convenience):

The following diagram contains level 2 without the dashed lines from level 1. Once this process has been repeated several times, the fractal will begin to look like one-half of a feather (peek ahead to Fig. 22.5 to see the fractal at various stages):

22.7.3 (Optional) Fractal App GUI

In this section and the next, we build a JavaFX Fractal app that displays the Lo Fractal— we do not present the example's `Application` subclass because it performs the same tasks demonstrated in Chapters 15–16 to load and display the app's FXML.

The following diagram contains the GUI with the fx:id values for the controls we access programmatically—like the Painter app in Section 16.3, this app uses a `BorderPane` layout with a white background color:

In the top area, we placed a `ToolBar` (located in the Scene Builder Library's Containers section). A `ToolBar` layout arranges its controls horizontally (by default) or vertically. Typically, you place `ToolBar`s at your GUI's edges, such as in a `BorderPane`'s top, left, bottom or right areas.

In the center, we placed a 400-by-480 pixel `Canvas` (from the Scene Builder Library's Miscellaneous section). `Canvas` is a `Node` subclass in which you can draw graphics using a `GraphicsContext` (both in the package `javafx.scene.canvas`). To size the `BorderPane` layout to its contents, we reset its Pref Width and Pref Height property values to `USE_COMPUTED_SIZE`.

`ToolBar` and Its Additional Controls

By default, the `ToolBar` you drag onto your layout has one `Button`. You can drag additional controls onto the `ToolBar` and, if necessary, remove the default `Button`. We added a `ColorPicker` control (with fx:id `colorPicker`), another `Button` and a `Label` (with fx:id `levelLabel`). For the two `Button`s and the `Label`, we set their text as shown in the preceding screen capture.

`ColorPicker`

A **`ColorPicker`** provides a predefined color-selecting GUI that, by default, enables the user to choose colors by color swatches (small squares of sample colors). Initially, the selected color is White. We'll programmatically set this to Blue in the app's controller. The following diagram contains the `ColorPicker`'s default GUI and the Custom Colors GUI that appears when you click the `ColorPicker`'s Custom Color… link. The Custom Colors GUI enables you to select any custom color.

Event Handlers

An `ActionEvent` occurs when the user selects a color swatch or customizes a color and presses Save in the Custom Colors GUI. In Scene Builder, for the `ColorPicker`'s On Action event handler, we specified `colorSelected`, and for the Decrease Level and Increase Level `Button`'s On Action events handlers, we specified `decreaseLevelButtonPressed` and `increaseLevelButtonPressed`, respectively.

22.7.4 (Optional) `FractalController` Class

Class `FractalController` (Fig. 22.5) defines the app's event handlers and the methods for drawing the Lo Fractal recursively. The sample screen captures contain the fractal for levels 0–5, 8 and 11. If we focus on one of the arms of this fractal, it will be similar to the whole image of the previous level. This property defines the fractal to be strictly self-similar.

Click here to view code image

```
1  // Fig. 22.5: FractalController.java
2  // Drawing the "Lo feather fractal" using recursion.
3  import javafx.event.ActionEvent;
4  import javafx.fxml.FXML;
```

```
5  import javafx.scene.canvas.Canvas;
6  import javafx.scene.canvas.GraphicsContext;
7  import javafx.scene.control.ColorPicker;
8  import javafx.scene.control.Label;
9  import javafx.scene.paint.Color;
10
11
12 public class FractalController {
13    // constants
14    private static final int MIN_LEVEL = 0;
15    private static final int MAX_LEVEL = 15;
16
17    // instance variables that refer to GUI components
18    @FXML private Canvas canvas;
19    @FXML private ColorPicker colorPicker;
20    @FXML private Label levelLabel;
21
22    // other instance variables
23    private Color currentColor = Color.BLUE;
24    private int level = MIN_LEVEL; // initial fractal level
25    private GraphicsContext gc; // used to draw on Canvas
26
27    // initialize the controller
28    public void initialize() {
29       levelLabel.setText("Level: " + level);
30       colorPicker.setValue(currentColor); // start with blue
31       gc = canvas.getGraphicsContext2D(); // get the GraphicsContext
32       drawFractal();
33    }
34
35    // sets currentColor when user chooses a new Color
36    @FXML
37    void colorSelected(ActionEvent event) {
38       currentColor = colorPicker.getValue(); // get new Color
39       drawFractal();
40    }
41
42    // decrease level and redraw fractal
43    @FXML
44    void decreaseLevelButtonPressed(ActionEvent event) {
45       if (level > MIN_LEVEL) {
46          --level;
47          levelLabel.setText("Level: " + level);
48          drawFractal();
49       }
50    }
51
52    // increase level and redraw fractal
53    @FXML
54    void increaseLevelButtonPressed(ActionEvent event) {
```

```
55          if (level < MAX_LEVEL) {
56             ++level;
57             levelLabel.setText("Level: " + level);
58             drawFractal();
59          }
60       }
61
62       // clear Canvas, set drawing color and draw the fractal
63       private void drawFractal() {
64          gc.clearRect(0, 0, canvas.getWidth(), canvas.getHeight());
65          gc.setStroke(currentColor);
66          drawFractal(level, 40, 40, 350, 350);
67       }
68
69       // draw fractal recursively
70       public void drawFractal(int level, int xA, int yA, int xB, int yB) {
71          // base case: draw a line connecting two given points
72          if (level == 0) {
73             gc.strokeLine(xA, yA, xB, yB);
74          }
75          else { // recursion step: determine new points, draw next level
76             // calculate midpoint between (xA, yA) and (xB, yB)
77             int xC = (xA + xB) / 2;
78             int yC = (yA + yB) / 2;
79
80             // calculate the fourth point (xD, yD) which forms an
81             // isosceles right triangle between (xA, yA) and (xC, yC)
82             // where the right angle is at (xD, yD)
83             int xD = xA + (xC - xA) / 2 - (yC - yA) / 2;
84             int yD = yA + (yC - yA) / 2 + (xC - xA) / 2;
85
86             // recursively draw the Fractal
87             drawFractal(level - 1, xD, yD, xA, yA);
88             drawFractal(level - 1, xD, yD, xC, yC);
89             drawFractal(level - 1, xD, yD, xB, yB);
90          }
91       }
92    }
```

Fig. 22.5 | Drawing the "Lo feather fractal" using recursion.

`FractalJPanel` Fields

Lines 14–25 declare the class's fields:

- The constants `MIN_LEVEL` and `MAX_LEVEL` (lines 14–15) specify the range of Lo Fractal levels this app can draw. Once you reach level 15, the changes in the fractal (at this size) are not noticeable, so we set 15 as the maximum level. Around

level 13 and higher, the fractal rendering may become slower due to the details to be drawn.

- Lines 18–20 declare the `@FXML` instance variables that will reference the GUI's controls we programmatically manipulate. Recall that these are initialized by the `FXMLLoader`.
- Line 23 declares the `Color` variable `currentColor` and initializes it to the constant `Color.BLUE`. Class **`Color`** (package `javafx.scene.paint`) contains various constants for common colors and provides methods for creating custom colors.
- Line 24 declares the `int` variable `level`, which maintains the current fractal level.
- Line 25 declares the `GraphicsContext` variable `gc`, which will be used to draw on the app's `Canvas`.

`initialize` Method

When the app's controller is created and initialized, method `initialize` (lines 28–33):

- Sets the `levelLabel`'s initial text to indicate level 0.
- Sets the initial value of the `colorPicker` to `currentColor` (`Color.BLUE`).
- Gets the `canvas`'s `GraphicsContext` that will be used to draw lines in the currently selected color.
- Calls `drawFractal` (lines 63–67) to draw level 0 of the fractal.

`colorSelected` Event Handler

When the user chooses a new color, the `colorPicker`'s `colorSelected` event handler (lines 36–40) uses `ColorPicker` method **`getValue`** to get the currently selected color, then calls `drawFractal` (lines 63–67) to redraw the fractal in the new color and current `level`.

`decreaseLevelButtonPressed` and `increaseLevelButtonPressed` Event Handlers

When the user presses the Decrease Level or Increase Level

`Button`s, the corresponding event handler (lines 43–50 or 53–60) executes. These event handlers decrease or increase the level, set the `levelLabel`'s text accordingly and call `drawFractal` (lines 63–67) to redraw the fractal for the new `level` and `currentColor`. Attempting to decrease the `level` below `MIN_LEVEL` or above `MAX_LEVEL` does nothing in this app.

`drawFractal` Method with No Arguments

Each time `drawFractal` is called, it uses `GraphicsContext` method **`clearRect`** to clear any prior drawing. This method clears a rectangular area by setting the contents of the specified area to the `Canvas`'s background color. The method's four arguments are the *x*- and *y*-coordinates of the rectangle's upper-left corner and the width and height of the rectangle. In this case, we clear the entire `Canvas`—methods **`getWidth`** and **`getHeight`** return the `Canvas`'s width and height, respectively.

Next, line 65 calls `GraphicsContext` method **`setStroke`** to set the drawing color to the `currentColor`. Then, line 66 makes the first call to the overloaded recursive method `drawFractal` (lines 70–91) to draw the fractal.

`drawFractal` Method with Five Arguments

Lines 70–91 define the recursive method that creates the fractal. This method takes five parameters: the level of the Lo Fractal to draw and four integers that specify the *x*- and *y*-coordinates of two points. The base case for this method (line 72) occurs when `level` equals `0`—line 73 uses `GraphicsContext` method **`strokeLine`** to draw a line between the two points the current call to `drawFractal` received as arguments.

In the recursive step (lines 75–90), lines 77–84 calculate

- (`xC`, `yC`)—the midpoint between (`xA`, `yA`) and (`xB`, `yB`), and
- (`xD`, `yD`)—the point that creates a right isosceles triangle with (`xA`, `yA`) and (`xC`, `yC`).

Then, lines 87–89 make three recursive calls on three different sets of points.

Since no lines will be drawn until the base case is reached, the distance between two points decreases on each recursive call. As the level of recursion increases, the fractal becomes smoother and more detailed. The shape of this fractal stabilizes as the level approaches

12—the shape of the fractal remains approximately the same, and the additional details become hard to perceive. Fractals will stabilize at different levels based on their shape and size.

Generative AI

1 Prompt genAIs for examples of fractals in nature.
2 Prompt genAIs for uses of fractals in mathematics and science.

22.8 Recursive Backtracking

Our recursive methods all have a similar architecture—if the base case is reached, return a result; if not, make one or more recursive calls. This section explores a more complex recursive technique that finds a path through a maze, returning true if there's a possible solution. The solution involves moving through the maze one step at a time, where moves can be made by going down, right, up or left (diagonal moves are prohibited). From the current location in the maze (starting with the entry point), the following steps are taken: For each possible direction, the move is made in that direction, and a recursive call is made to solve the remainder of the maze from the new location. When a dead end is reached (i.e., we cannot take any more steps forward without hitting a wall), we back up to the previous location and try to go in a different direction. If no other direction can be taken, we back up again. This process continues until we find a point in the maze where a move can be made in another direction. Once such a location is found, we move in the new direction and continue with another recursive call to solve the rest of the maze.

To back up to the previous location in the maze, our recursive method simply returns false, moving up the method-call chain to the previous recursive call (which references the previous location in the maze). Using recursion to return to an earlier decision point is known as **recursive backtracking**. If one set of recursive calls does not result in a solution to the problem, the program backs up to the previous decision point and makes a different decision, often resulting in another set of recursive calls. In this example, the previous decision point is the previous location in the maze, and the decision to be made is the direction the next move should take. One direction has led to a dead end, so the search continues with a different direction. The recursive backtracking solution to the maze problem uses recursion to return only partway up the method-call

chain then try a different direction. If the backtracking reaches the entry location of the maze and the paths in all directions have been attempted, the maze does not have a solution.

Generative AI

1 Prompt genAIs for real-world applications of recursive backtracking.

22.9 Big *O* Notation

One way to describe an algorithm's effort is with **Big *O* notation**, which indicates how hard an algorithm may have to work to solve a problem. The Big *O* for searching and sorting algorithms depends mainly on how many data elements there are. We use Big *O* to describe various sorting algorithms' worst-case run times.

22.9.1 *O*(1) Algorithms

Suppose an algorithm tests whether an array's first element is equal to its second. If the array has 10 elements, this algorithm requires one comparison. If the array has 1,000 elements, the algorithm still requires one comparison. In fact, the algorithm is completely independent of the array's number of elements. This algorithm is said to have **constant run time**, which we represent in Big *O* notation as ***O*(1)** and pronounce "order 1." An *O*(1) algorithm does not necessarily require only one comparison. *O*(1) means that the number of comparisons is **constant**—it does not grow as the array size increases. An algorithm that tests whether the first array element is equal to any of the next three elements is still *O*(1), even though it requires three comparisons.

22.9.2 *O*(*n*) Algorithms

An algorithm that tests whether an array's first element is equal to any of the array's other elements requires at most $n - 1$ comparisons, where n is the array's number of elements. If the array has 10 elements, this algorithm requires up to nine comparisons. If the array has 1,000 elements, this algorithm requires up to 999 comparisons.

As n grows, the n in the expression $n - 1$ "dominates," so

subtracting 1 becomes inconsequential. Big O is designed to highlight these dominant terms and ignore those that become unimportant as n grows. For this reason, an algorithm that requires n –1 comparisons is said to be **$O(n)$**. An $O(n)$ algorithm is referred to as having a **linear run time**. $O(n)$ is often pronounced "on the order of n" or just "order n."

22.9.3 $O(n^2)$ Algorithms

Suppose an algorithm tests whether any array element is duplicated elsewhere in the array. The algorithm compares the first element with all of the array's other elements. It then compares the second element with all of the array's other elements except the first—the second was already compared to the first. Then, the algorithm compares the third element with all the other elements except the first two. Ultimately, this algorithm makes a total of $(n - 1) + (n - 2) + \ldots + 2 + 1$ or $n^2/2 - n/2$ comparisons. As n increases, the n^2 term dominates, and the n term becomes inconsequential. Big O notation highlights the n^2 term, leaving $n^2/2$. Constant factors (such as 1/2) are omitted in Big O notation.

Big O is concerned with how an algorithm's run time grows in relation to the number of items processed. Suppose an algorithm requires n^2 comparisons. With four elements, the algorithm will require 16 comparisons; with eight elements, the algorithm will require 64 comparisons. With this algorithm, doubling the number of elements **quadruples** the number of comparisons. Consider a similar algorithm requiring $n^2/2$ comparisons. With four elements, the algorithm will require eight comparisons; with eight elements, the algorithm will require 32 comparisons. Again, doubling the number of elements **quadruples** the number of comparisons. Both algorithms grow as the square of n, so Big O ignores the constant and both algorithms are considered to be **$O(n^2)$**. This is called **quadratic run time** and pronounced "on the order of n-squared" or simply "order n-squared."

When n is small, $O(n^2)$ algorithms (running on today's billion-operations-per-second personal computers) will not noticeably affect performance. But as n grows, you'll start to notice the performance degradation. An $O(n^2)$ algorithm running on a million-element array would require a trillion "operations," where each could require several machine instructions. This could require a few hours to execute. A billion-element array would require a quintillion operations, a number so large that the algorithm could

take decades! $O(n^2)$ algorithms are often easy to write. We'll also present an algorithm with a more favorable Big O measure. Efficient algorithms require clever coding and more work to create. Their superior performance can be worth the extra effort, especially as n gets large.

Generative AI

1 Prompt genAIs, asking about the origin of Big O notation.
2 Prompt genAIs, asking for real-world examples of how Big O affects application performance.

22.10 Common Big O Notations

The following table lists various common Big O categories and several values for n to highlight the differences in the growth rates. If you interpret the values in the table as seconds of calculation, you can easily see why $O(n^2)$ algorithms should be avoided!

$O(\log n)$	$O(\log n)$
0	0
2	2
3	3
4	46
5	55
10	100
20	200,000
30	$3{,}000^6$
60	$6{,}000{,}000^{12}$
90	$9{,}000{,}000{,}000^{18}$

22.11 Wrap-Up

In this chapter, you created recursive methods that call themselves. You divided each problem into two conceptual pieces—the base case that the method knows how to do and the recursion step that is a slightly smaller version of the original problem and is performed by a recursive method call. We presented some popular recursion examples, including calculating factorials and producing values in the Fibonacci series. We discussed the order in which recursive method calls are pushed on or popped off the program-execution

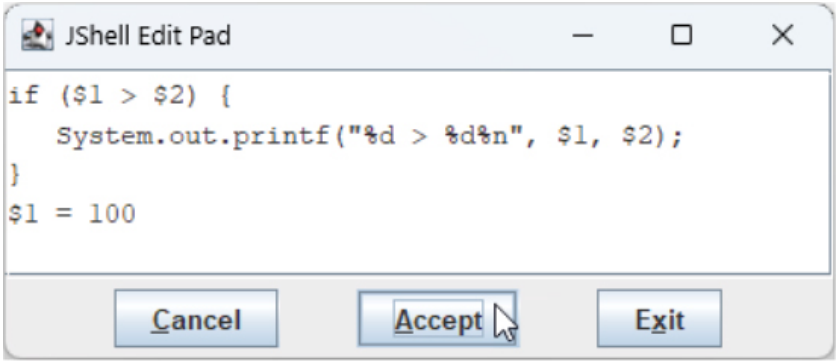
stack. Next, we compared recursive and iterative approaches. We used recursion to solve more complex problems—the Towers of Hanoi and displaying fractals. We discussed recursive backtracking—a technique for solving problems that involves backing up through recursive calls to try different possible solutions.

This concludes *Java for Programmers with Generative AI, 5/e*. We hope you've found this modern computer programming presentation informative and entertaining. We wish you great success!

JShell Edit Pad

```
if ($1 < $2) {
   System.out.printf("%d < %d%n", $1, $2);
}
```

Cancel Accept Exit



Introduction to JShell for Interactive Java

Objectives

In this appendix, you'll:

- Use JShell to enhance the learning and software development processes by quickly exploring, discovering and experimenting with Java language and API features.
- Start a JShell session.
- Execute code snippets.
- Declare variables.
- Evaluate expressions.

- Edit existing code snippets.
- Declare and use a class.
- Save snippets to a file.
- Open a file of JShell snippets and evaluate them.
- Auto-complete code and JShell commands.
- Display method parameters and overloads.
- Use the Java API documentation in JShell.
- Declare and use methods.
- Forward reference a method that has not yet been declared.
- Learn how JShell wraps exceptions.
- Import custom packages for use in a JShell session.
- Control JShell's feedback level.

Outline

A.1 Introduction

A.2 Introduction to JShell

A.2.1 Starting a JShell Session

A.2.2 Executing Statements

A.2.3 Declaring Variables Explicitly

A.2.4 Listing and Executing Prior Snippets

A.2.5 Evaluating Expressions and Declaring Variables Implicitly

A.2.6 Using Implicitly Declared Variables

A.2.7 Viewing a Variable's Value

A.2.8 Resetting a JShell Session

A.2.9 Writing Multiline Statements

A.2.10 Editing Code Snippets

A.2.11 Exiting JShell

A.3 Command-Line Input in JShell

A.4 Declaring and Using Classes

A.4.1 Creating a Class in JShell

A.4.2 Explicitly Declaring Reference-Type Variables

A.4.3 Creating Objects

A.4.4 Manipulating Objects

A.4.5 Creating a Meaningful Variable Name for an Expression

A.4.6 Saving and Opening Code-Snippet Files
A.5 Discovery with JShell Auto-Completion
A.5.1 Auto-Completing Identifiers
A.5.2 Auto-Completing JShell Commands
A.6 Exploring a Class's Members and Viewing Documentation
A.6.1 Listing Class `Math`'s `static` Members
A.6.2 Viewing a Method's Parameters
A.6.3 Viewing a Method's Documentation
A.6.4 Viewing a `public` Field's Documentation
A.6.5 Viewing a Class's Documentation
A.6.6 Viewing Method Overloads
A.6.7 Exploring Members of a Specific Object
A.7 Declaring Methods
A.7.1 Forward Referencing an Undeclared Method—Declaring Method `displayCubes`
A.7.2 Declaring a Previously Undeclared Method
A.7.3 Testing `cube` and Replacing Its Declaration
A.7.4 Testing Updated Method `cube` and Method `displayCubes`
A.8 Exceptions
A.9 Importing Classes and Adding Packages to the `CLASSPATH`
A.10 Using an External Editor
A.11 Summary of JShell Commands
A.11.1 Getting Help in JShell
A.11.2 `/edit` Command: Additional Features
A.11.3 `/reload` Command
A.11.4 `/drop` Command
A.11.5 Feedback Modes
A.11.6 Other JShell Features Configurable with `/set`
A.12 Keyboard Shortcuts for Snippet Editing
A.13 How JShell Reinterprets Java for Interactive Use
A.14 IDE JShell Support
A.15 Wrap-Up
Self-Review Exercises with Answers

A.1 Introduction

Instructors have indicated a preference in introductory programming courses for languages with **REPLs (read-evaluate-print loops)**. Java's **JShell** provides a rich REPL experience. Third parties have built JShell and related interactive-development tools into major IDEs like IntelliJ, Eclipse, NetBeans and others.

What is JShell?

What's the magic? It's simple. JShell provides a fast and friendly environment that lets you quickly explore, discover and experiment with Java language features and its extensive libraries. REPLs like the one in JShell have been around for decades. In the 1960s, one of the earliest REPLs made convenient interactive development possible in the LISP programming language. Students of that era, like one of your authors, Harvey Deitel, found it fast and fun to use.

JShell replaces the editing, compiling and executing cycle with its read-evaluate-print loop. Rather than complete programs, you write **JShell commands** and Java code snippets. When you enter a snippet, JShell immediately **reads** it, **evaluates** it and **prints** the results that help you observe the effects of your code. Then, it **loops** to perform this process again for the next snippet. As you work through the scores of examples and exercises, you'll learn how JShell and its instant feedback keep your attention, enhance your performance, and speed the learning and software development processes.

Code Comes Alive

As you know, we emphasize the value of the live-code teaching approach in our books, focusing on complete, working programs. JShell brings this to the individual snippet level. Your code literally comes alive as you enter each line. Of course, you'll still make occasional errors as you enter your snippets. JShell reports compilation errors to you on a snippet-by-snippet basis. You can use this capability, for example, to test common programming errors and observe the errors as they occur.

Kinds of Snippets

Snippets can be expressions, individual statements, multi-line statements and larger entities, like methods and classes. JShell supports all but a few Java features, but there are some differences designed to facilitate JShell's explore–discover–experiment capabilities. In JShell, methods do not need to be in classes, expressions and statements do not need to be in methods, and you do not need a `main` method (other differences are in Section A.13). Eliminating this infrastructure saves you considerable time, especially compared to the lengthy repeated edit, compile and execute cycles of complete programs. And because JShell automatically displays the results of evaluating your expressions and statements, you do not need as many print statements as we use throughout this book's traditional Java code examples.

Discovery with Auto-Completion

We include a detailed treatment of **auto-completion**—a key discovery feature that speeds the coding process. After you enter a portion of a name (class, method, variable, etc.) and press (or select) the *Tab* key, JShell completes the name for you or provides a list of all possible names that begin with what you've entered so far. You can then easily display method parameters and the documentation that describes those methods.

Rapid Prototyping

Professional developers commonly use JShell for experimentation and rapid prototyping but not full-out software development. Once you create and test a small chunk of code, you can paste it into your larger project.

How This Appendix Is Organized

This appendix is optional. For those who want to use JShell, the appendix is designed as a series of units, paced to specific earlier chapters. Each unit begins with a statement like "This section may be read after studying Chapter 2." So you'd start by reading through Chapter 2, then read this appendix's corresponding section—and similarly for subsequent chapters.

Chapter 2 JShell Self-Review Exercises with Answers

As you work through this appendix, execute each snippet and command in JShell to confirm that the features work as advertised.

Sections A.2–A.3 are designed to be read after Chapter 2. Once you read these sections, we recommend doing the dozens of self-review exercises at the end of this appendix. JShell encourages you to "learn by doing," so the exercises have you write and test code snippets that exercise many of Chapter 2's Java features. The self-review exercises are small and to the point, and each is followed immediately by its answer to help you quickly get comfortable with JShell's capabilities.

A.2 Introduction to JShell

[This section may be read after studying Chapter 2, Intro to Java Programming.]

In Chapter 2, to create a Java application, you

- created a class containing a `main` method,
- declared in `main` the statements to execute when you run the program,
- compiled the program and fixed any compilation errors that occurred—his step had to be repeated until the program compiled without errors, and
- ran the program to observe the results.

By automatically compiling and executing code as you complete each expression or statement, JShell eliminates the overhead of

- creating a class containing the code you wish to test,
- compiling the class and
- executing the class.

Instead, you can focus on interactively discovering and experimenting with Java's language and API features. If you enter code that does not compile, JShell immediately reports the errors. You can then use JShell's editing features to quickly fix and re-execute the code.

A.2.1 Starting a JShell Session

To start a JShell session, execute the **`jshell` command** in your system's command line:

- On Microsoft Windows, open a Command Prompt or PowerShell window.
- On macOS, open a Terminal window.

This starts a new JShell session, displaying the following message and the **`jshell>` prompt**:

Click here to view code image

```
| Welcome to JShell -- Version 23
| For an introduction type: /help intro

jshell>
```

The first line above indicates that you're using `Version 23` of the JDK. JShell precedes informational messages with vertical bars (`|`). You are now ready to enter Java code or JShell commands.

A.2.2 Executing Statements

[As you work through this appendix, enter the same code and JShell commands that we present in each `jshell>` prompt to ensure that your outputs will match ours.]

JShell has two input types:

- Java code, which the JShell documentation refers to as **snippets** and
- JShell commands.

In this section and Section A.2.3, we begin with Java code snippets. Subsequent sections introduce JShell commands.

You can enter any expression or statement at the `jshell>` prompt, then press (or select) *Enter* to execute the code and observe its results immediately. Consider the program of Fig. 2.1, which we list again in Fig. A.1. To demonstrate how `System.out.println` works, this program required many lines of code and comments, which you had to write, compile and execute. Even without the comments, five code lines were still needed (lines 4 and 6–9).

Click here to view code image

```
1  // Fig. A.1: Welcome1.java
2  // Text-printing program.
```

```
3
4  public class Welcome1 {
5     // main method begins program execution
6     public static void main(String[] args) {
7        System.out.println("Welcome to Java Programming!");
8     } // end method main
9  } // end class Welcome1
```

```
Welcome to Java Programming!
```

Fig. A.1 | Text-printing program.

In JShell, you can execute the statement in line 7 without creating all the infrastructure of class `Welcome1` and its `main` method:

Click here to view code image

```
jshell> System.out.println("Welcome to Java Programming!")
Welcome to Java Programming!

jshell>
```

In this case, JShell displays the snippet's command-line output below the initial `jshell>` prompt and the statement you entered. Per our convention, we use **`bold`** for user inputs.

We did not enter the statement's semicolon (`;`). JShell adds terminating semicolons only for standalone statements.[1] For example, if the statement is inside braces (`{` and `}`), you must include the semicolon.

1. Not requiring semicolons is one example of how JShell reinterprets standard Java for convenient interactive use. We discuss several of these throughout the appendix and summarize them in Section A.13.

The blank line before the second `jshell>` prompt results from the newline displayed by method `println` and a separate newline that JShell always displays before each `jshell>` prompt. Using `print` rather than `println` eliminates the blank line:

Click here to view code image

```
jshell> System.out.print("Welcome to Java Programming!")
Welcome to Java Programming!
jshell>
```

JShell keeps track of everything you enter, which can be useful for re-executing prior statements and modifying statements to update

the tasks they perform.

A.2.3 Declaring Variables Explicitly

Almost anything you can declare in a typical Java source-code file can also be declared in JShell (Section A.13 discusses some of the features you cannot use). For example, you can explicitly declare a variable as follows:

```
jshell> int number1
number1 ==> 0

jshell>
```

When you enter a variable declaration, JShell displays the variable's name (in this case, `number1`) followed by `==>` (which means "has the value") and the variable's initial value (`0`). If you do not specify an initial value explicitly, the variable is initialized to its type's default value—in this case, `0` for an `int` variable.

A variable can be initialized in its declaration—let's redeclare `number1`:

```
jshell> int number1 = 30
number1 ==> 30

jshell>
```

JShell displays

```
number1 ==> 30
```

to indicate that `number1` now has the value `30`. When you declare a new variable with the same name as another variable in the current JShell session, JShell replaces the first declaration with the new one.[2] Because `number1` was declared previously, we could have simply assigned `number1` a value, as in

2. Redeclaring an existing variable is another example of how JShell reinterprets standard Java for interactive use. This behavior differs from how the Java compiler handles a new declaration of an existing variable—such a "double declaration" generates a compilation error.

```
jshell> number1 = 45
number1 ==> 45

jshell>
```

Compilation Errors in JShell

You must declare variables before using them in JShell. The following declaration of `int` variable `sum` attempts to use a variable named `number2` that we have not yet declared, so JShell reports a compilation error, indicating that the compiler was unable to find a variable named `number2`:

Click here to view code image

```
jshell> int sum = number1 + number2
|  Error:
|  cannot find symbol
|    symbol:   variable number2
|  int sum = number1 + number2;
|                      ^-----^

jshell>
```

The error message uses the notation `^-----^` to highlight the error in the statement. No error is reported for the previously declared variable `number1`. This snippet has a compilation error, so it's invalid. However, JShell still maintains the snippet as part of the JShell session's history, which includes valid snippets, invalid snippets and the commands you've entered. As you'll soon learn, you can recall this invalid snippet and execute it again later. JShell's **`/history` command** displays the current session's history—that is, everything you've entered:

Click here to view code image

```
jshell> /history

System.out.println("Welcome to Java Programming!")
System.out.print("Welcome to Java Programming!")
int number1
int number1 = 30
number1 = 45
int sum = number1 + number2
/history

jshell>
```

Fixing the Error

JShell makes it easy to fix a prior error and re-execute a snippet. Let's fix the preceding error by first declaring `number2` with the value `72`:

```
jshell> int number2 = 72
number2 ==> 72

jshell>
```

Subsequent snippets can now use `number2`—in a moment, you'll re-execute the snippet that declared and initialized `sum` with `number1 + number2`.

Recalling and Re-executing a Previous Snippet

Now that `number1` and `number2` are declared, we can declare the `int` variable `sum`. You can use the up and down arrow keys to navigate backward and forward through the snippets and JShell commands you've entered previously. Rather than re-entering `sum`'s declaration, you can press (or select) the up arrow key three times to recall the declaration that failed previously. JShell recalls your prior inputs in reverse order—the last snippet or command you entered is recalled first. So, the first time you press (or select) the up arrow key, the following appears at the `jshell>` prompt:

```
jshell> int number2 = 72
```

The second time you press (or select) the up arrow key, the `/history` command appears:

```
jshell> /history
```

The third time you press (or select) the up arrow key, `sum`'s prior declaration appears:

Click here to view code image

```
jshell> int sum = number1 + number2
```

Now you can press (or select) *Enter* to re-execute the snippet that declares and initializes `sum`:

Click here to view code image

```
jshell> int sum = number1 + number2
sum ==> 117

jshell>
```

JShell adds the values of `number1` (`45`) and `number2` (`72`), stores the result in the new `sum` variable, then outputs `sum`'s value (`117`).

A.2.4 Listing and Executing Prior Snippets

You can view a list of all previous valid Java code snippets with JShell's **`/list` command**—JShell displays the snippets in the order you entered them:

Click here to view code image

```
jshell> /list

   1 : System.out.println("Welcome to Java Programming!")
   2 : System.out.print("Welcome to Java Programming!")
   4 : int number1 = 30;
   5 : number1 = 45
   6 : int number2 = 72;
   7 : int sum = number1 + number2;

jshell>
```

Each valid snippet is identified by a sequential **snippet ID**. The snippet with ID 3 is missing above because we replaced that original variable-declaration snippet

```
int number1
```

with the one that has the ID 4 in the preceding `/list`. The `/list` command does not display everything that `/history` does. As you recall, if you omit a terminating semicolon, JShell inserts it for you behind the scenes. When you execute `/list`, only the declarations (snippets 4, 6 and 7) include the semicolons that JShell inserted. Snippet 1 above is just an expression. If we enter it with a terminating semicolon, it's an **expression statement**.

Executing Snippets By ID Number

You can execute any prior snippet by entering `/`***id***, where *id* is the snippet's ID. For example, when you enter `/1`:

Click here to view code image

```
jshell> /1
System.out.println("Welcome to Java Programming!")
Welcome to Java Programming!

jshell>
```

JShell displays the current session's snippet with the ID `1`, executes it and displays the result.

Re-executing Several Individual Snippets or Snippet Ranges

You can also re-execute more than one statement at a time. For example, the following would re-execute the range of snippets from *start* through *end*, inclusive:

```
/start-end
```

and the following would re-execute separate individual snippets or snippet ranges:

```
/id1 id2 id3 ...
```

In this case, the IDs can be can be individual snippet IDs or ID ranges.

Re-executing the Last Snippet

To re-execute the previous snippet, use `/!`:

Click here to view code image

```
jshell> /!
System.out.println("Welcome to Java Programming!")
Welcome to Java Programming!

jshell>
```

JShell assigns an ID to every valid snippet you execute, so even though

Click here to view code image

```
System.out.println("Welcome to Java Programming!")
```

already exists in this session as snippet `1`, JShell creates a new snippet with the next ID in sequence (in this case, `8` and `9` for the last two snippets). The `/list` command's output confirms that snippets `1`, `8` and `9` are identical:

Click here to view code image

```
jshell> /list

   1 : System.out.println("Welcome to Java Programming!")
   2 : System.out.print("Welcome to Java Programming!")
   4 : int number1 = 30;
```

```
   5 : number1 = 45
   6 : int number2 = 72;
   7 : int sum = number1 + number2;
   8 : System.out.println("Welcome to Java Programming!")
   9 : System.out.println("Welcome to Java Programming!")

jshell>
```

A.2.5 Evaluating Expressions and Declaring Variables Implicitly

When you enter an expression in JShell, it evaluates the expression, implicitly creates a variable and assigns the expression's value to the variable. **Implicit variables** are named `$#`, where # is the new snippet's ID.[3] For example:

```
jshell> 11 + 5
$10 ==> 16

jshell>
```

3. Implicitly declared variables are another example of how JShell reinterprets standard Java for interactive use. In regular Java programs, you must explicitly declare *every* variable.

evaluates the expression `11 + 5` and assigns the resulting value (`16`) to the implicitly declared variable `$10` because there were nine prior valid snippets (even though one was deleted because we redeclared the variable `number1`). JShell infers the type of `$10` as `int` because the expression `11 + 5` adds two `int` values, producing an `int`. Expressions may also include one or more method calls. The list of snippets is now:

Click here to view code image

```
jshell> /list

   1 : System.out.println("Welcome to Java Programming!")
   2 : System.out.print("Welcome to Java Programming!")
   4 : int number1 = 30;
   5 : number1 = 45
   6 : int number2 = 72;
   7 : int sum = number1 + number2;
   8 : System.out.println("Welcome to Java Programming!")
   9 : System.out.println("Welcome to Java Programming!")
  10 : 11 + 5
```

```
jshell>
```

Note that the implicitly declared variable `$10` appears in the list simply as `10` without the `$`.

A.2.6 Using Implicitly Declared Variables

You can use implicitly declared variables in expressions. For example, the following assigns to the existing variable `sum` the result of adding `number1` (`45`) and `$10` (`16`):

```
jshell> sum = number1 + $10
sum ==> 61

jshell>
```

The list of snippets is now:

Click here to view code image

```
jshell> /list

   1 : System.out.println("Welcome to Java Programming!")
   2 : System.out.print("Welcome to Java Programming!")
   4 : int number1 = 30;
   5 : number1 = 45
   6 : int number2 = 72;
   7 : int sum = number1 + number2;
   8 : System.out.println("Welcome to Java Programming!")
   9 : System.out.println("Welcome to Java Programming!")
  10 : 11 + 5
  11 : sum = number1 + $10

jshell>
```

A.2.7 Viewing a Variable's Value

You can view a variable's value at any time by entering its name and pressing *Enter*:

```
jshell> sum
sum ==> 61

jshell>
```

Doing this does not create another implicitly declared variable.

A.2.8 Resetting a JShell Session

You can remove all prior code from a JShell session by entering the **`/reset` command**:

```
jshell> /reset
|  Resetting state.

jshell> /list

jshell>
```

The subsequent `/list` command has no output, confirming that all prior snippets were removed. Messages displayed by JShell, such as

```
| Resetting state.
```

are helpful when you're becoming familiar with JShell. In Section A.11.5, we'll demonstrate how you can change the JShell feedback mode, making it more or less verbose.

A.2.9 Writing Multiline Statements

Next, let's enter an `if` statement that determines whether `45` is less than `72`. First, store 45 and 72 in implicitly declared variables, as in:

```
jshell> 45
$1 ==> 45

jshell> 72
$2 ==> 72

jshell>
```

Next, begin entering the `if` statement:

```
jshell> if ($1 < $2) {
   ...>     _
```

JShell knows the `if` statement is incomplete because we entered the opening left brace but did not provide a body or a closing right brace. So, JShell displays the **continuation prompt** `...>` at which you can enter more of the control statement. It also indents the second line four spaces. We used an underscore above to indicate where the cursor currently resides.[4] The following completes and

evaluates the `if` statement:

4. Java programmers commonly use four-space indents.

Click here to view code image

```
jshell> if ($1 < $2) {
   ...>     System.out.printf("%d < %d%n", $1, $2);
   ...> }
45 < 72

jshell>
```

The second continuation prompt appeared because the `if` statement was still missing its closing right brace (`}`). When you enter the closing brace, JShell automatically aligns it under the `"i"` in the `"if"` keyword. The statement-terminating semicolon (`;`) at the end of the `System.out.printf` statement in the `if`'s body is required. JShell assigns each multiline code snippet—such as an `if` statement—one snippet ID. The list of snippets is now:

Click here to view code image

```
jshell> /list

   1 : 45
   2 : 72
   3 : if ($1 < $2) {
           System.out.printf("%d < %d%n", $1, $2);
       }

jshell>
```

A.2.10 Editing Code Snippets

Sometimes you might want to create a new snippet based on an existing one in the current JShell session. For example, suppose you want to create an `if` statement that determines whether `$1` is greater than `$2`. The statement that performs this task

Click here to view code image

```
if ($1 > $2) {
   System.out.printf("%d > %d%n", $1, $2);
}
```

is nearly identical to the `if` statement in Section A.2.9, so it would be easier to edit the existing statement rather than enter the new one from scratch. When you edit a snippet, JShell saves the edited version as a new snippet with the next snippet ID in sequence.

Editing a Single-Line Snippet

To edit a snippet:

- locate it with the up-arrow key,
- use the arrow keys to navigate to the location in the snippet requiring changes,
- make your changes within the snippet, then
- press (or select) *Enter* to evaluate it.

Section A.12 lists keyboard shortcuts that can help you edit snippets.

Editing a Multiline Snippet

For a larger multiline snippet—such as an `if` statement—you can also edit the entire snippet by using JShell's **`/edit` command** to open the snippet in the JShell Edit Pad. The command

```
/edit
```

opens JShell Edit Pad and displays all valid code snippets you've entered so far. To edit a specific snippet, include the snippet ID, as in

```
/edit id
```

So, the command:

```
/edit 3
```

displays the `if` statement from Section A.2.9 in JShell Edit Pad: You can also edit groups and ranges of snippets, as discussed in Section A.2.4. JShell Edit Pad does not display snippet IDs. This window is modal—that is, while it's open, you cannot enter code snippets or commands at the JShell prompt until you dismiss the JShell Edit Pad window.

JShell Edit Pad supports only basic editing capabilities. You can:

- click to insert the cursor at a specific position to begin entering text,
- move the cursor via the arrow keys on your keyboard,
- drag the mouse to select text,
- use the *Delete* (*Backspace*) key to delete text,
- cut, copy and paste text using your operating system's keyboard shortcuts, and
- enter text, including new snippets separate from the one(s) you're editing.

In the first and second lines of the `if` statement, select each less than operator (`<`) and change it to a greater than (`>`) operator, then click Accept to create a new `if` statement containing the edited code. When you click Accept, JShell also immediately evaluates the new `if` statement and displays its results (if any)—`$1` (`45`) is not greater than `$2` (`72`), so the `System.out.printf` statement does not execute.[5]

5. We could have made this an `if`...`else` statement to output something when the condition is *false* but this section is meant to be used with Chapter 2 where we introduce only the single-selection `if` statement.

If you want to return immediately to the JShell prompt, rather than clicking Accept, you could click Exit to execute the edited snippet and close JShell Edit Pad. Clicking Cancel closes JShell Edit Pad and discards any changes you made since the last time you clicked Accept or since JShell Edit Pad was launched, if you have not yet clicked Accept.

When you change or create multiple snippets, then click Accept or Exit, JShell compares the JShell Edit Pad contents with the previously saved snippets. It then executes every modified or new snippet.

Adding a New Snippet Via JShell Edit Pad

To demonstrate that JShell Edit Pad does, in fact, execute snippets immediately when you click Accept, let's change `$1`'s value to `100` by entering the following statement after the `if` statement in JShell Edit Pad:

```
$1 = 100
```

and clicking Accept:

Each time you modify a variable's value, JShell immediately displays the variable's name and new value:

```
jshell> /edit 3
$1 ==> 100
```

Click Exit to close JShell Edit Pad and return to the `jshell>` prompt.

The following lists the current snippets—notice that each multiline `if` statement has only one ID:

Click here to view code image

```
jshell> /list

   1 : 45
   2 : 72
   3 : if ($1 < $2) {
           System.out.printf("%d < %d%n", $1, $2);
       }
   4 : if ($1 > $2) {
           System.out.printf("%d > %d%n", $1, $2);
       }
   5 : $1 = 100

jshell>
```

Executing the New `if` Statement Again

The following re-executes the new `if` statement (ID 4) with the updated `$1` value:

Click here to view code image

```
jshell> /4
if ($1 > $2) {
   System.out.printf("%d > %d%n", $1, $2);
}
100 > 72

jshell>
```

The condition `$1 > $2` is now `true`, so the `if` statement's body executes. The list of snippets is now

Click here to view code image

```
jshell> /list
```

```
1 : 45
2 : 72
3 : if ($1 < $2) {
        System.out.printf("%d < %d%n", $1, $2);
    }
4 : if ($1 > $2) {
        System.out.printf("%d > %d%n", $1, $2);
    }
5 : $1 = 100
6 : if ($1 > $2) {
        System.out.printf("%d > %d%n", $1, $2);
    }

jshell>
```

A.2.11 Exiting JShell

To terminate the current JShell session, use the **`/exit` command** or enter the keyboard shortcut *Ctrl* + *d* (or *control* + *d*).

A.3 Command-Line Input in JShell

[This section may be read after studying Chapter 2, Intro to Java Programming, and the preceding sections in this appendix.]

In Chapter 2, we performed command-line input using a `Scanner` object:

Click here to view code image

```
Scanner input = new Scanner(System.in);

System.out.print("Enter first integer: ");
int number1 = input.nextInt();
```

We created a `Scanner`, prompted the user for input, then used `Scanner` method `nextInt` to read a value. Recall that the program then waited for you to enter an integer and press (or select) *Enter* before proceeding to the next statement. The on-screen interaction appeared as follows:

```
Enter first integer: 45
```

This section demonstrates what that interaction looks like in JShell.

Creating a `Scanner`

Start a new JShell session or `/reset` the current one, then create a `Scanner` object:

Click here to view code image

```
jshell> Scanner input = new Scanner(System.in)
input ==> java.util.Scanner[delimiters=\p{javaWhitespace}+] ...
\E][infinity string=\Q∞\E]

jshell>
```

You do not need to import `Scanner`. JShell automatically imports the `java.util` package and several others—Section A.9 presents the complete list. When you create an object, JShell displays its text representation. The notation to the right of `input ==>` is the `Scanner`'s text representation (which you can simply ignore).

Prompting for Input and Reading a Value

Next, prompt the user for input:

Click here to view code image

```
jshell> System.out.print("Enter first integer: ")
Enter first integer:
jshell>
```

The statement's output is displayed immediately, followed by the next `jshell>` prompt. Now enter the input statement:

Click here to view code image

```
jshell> int number1 = input.nextInt()
_
```

At this point, JShell waits for your input. The input cursor is positioned below the `jshell>` prompt and snippet you just entered —indicated by the underscore (_) above— rather than next to the prompt `"Enter first integer:"` as it was in Chapter 2. Now enter an integer and press (or select) *Enter* to assign it to `number1`—the last snippet's execution is now complete, so the next `jshell>` prompt appears.:

Click here to view code image

```
jshell> int number1 = input.nextInt()
45
number1 ==> 45

jshell>
```

Though you can use `Scanner` for command-line input in JShell, it's awkward, and in most cases, unnecessary. The goal of the preceding interactions was to store an `int` in the variable `number1`. You can accomplish that in JShell with the simple assignment

```
jshell> int number1 = 45
number1 ==> 45

jshell>
```

So, you'll typically use assignments, rather than command-line input in JShell.

A.4 Declaring and Using Classes

[This section may be read after studying Chapter 8, Real-World Modeling with Custom Classes.]

Section A.2 demonstrated basic JShell features. Here, we create a class and manipulate an object of a class. We'll use the version of class `Account` presented in Fig. 8.1.

A.4.1 Creating a Class in JShell

Start a new JShell session (or `/reset` the current one), then declare class `Account`—we ignored the comments from Fig. 8.1:

Click here to view code image

```
jshell> public class Account {
   ...>    private String name;
   ...>
   ...>    public void setName(String name) {
   ...>       this.name = name;
   ...>    }
   ...>
   ...>    public String getName() {
   ...>       return name;
```

```
...>     }
...> }
|  created class Account

jshell>
```

JShell recognizes when you enter the class's closing brace—then displays

```
| created class Account
```

and issues the next `jshell>` prompt. Note that the semicolons throughout class `Account`'s body are required.

To save time, rather than entering a class's code, you can load an existing source code file into JShell, as we demonstrate in Section A.4.6. Though you can specify access modifiers like `public` on your classes (and other types), JShell ignores all access modifiers on the top-level types—that is, types that are not nested in other types.

Viewing Declared Classes

To view the names of the classes you've declared so far, enter the **`/types` command**:[6]

```
jshell> /types
|    class Account

jshell>
```

6. `/types` displays all types you declare, including `class`es, `record`s, `interface`s and `enum`s.

A.4.2 Explicitly Declaring Reference-Type Variables

The following creates the `Account` variable `account`:

```
jshell> Account account
account ==> null

jshell>
```

The default value of a reference-type variable is `null`.

A.4.3 Creating Objects

You can create new objects. The following creates an `Account` variable named `account` and initializes it with a new object:

Click here to view code image

```
jshell> account = new Account()
account ==> Account@2d928643

jshell>
```

The strange notation

```
Account@2d928643
```

is the default text representation of the new `Account` object. If a class provides a custom `toString` method, the text representation it returns will be output instead. The value after the `@` symbol is the object's hashcode.

Declaring an Implicit `Account` Variable Initialized with an `Account` Object

If you create an object with only the expression `new Account()`, JShell assigns the object to an implicit variable of type `Account`, as in:

```
jshell> new Account()
$4 ==> Account@49993335

jshell>
```

Note that this object's hashcode (`49993335`) differs from the prior `Account` object's hash-code (`2d928643`)—these typically are different, but that's not guaranteed.

Viewing Declared Variables

You can view all the variables you've declared so far with the JShell **`/vars` command**:

Click here to view code image

```
jshell> /vars
|    Account account = Account@2d928643
|    Account $4 = Account@49993335
```

```
jshell>
```

For each variable, JShell outputs the type and variable name followed by an equal sign and the variable's text representation.

A.4.4 Manipulating Objects

Once you have an object, you can call its methods. In fact, you already did this with the `System.out` object by calling its `println`, `print` and `printf` methods in earlier snippets.

The following sets the `account` object's name:

Click here to view code image

```
jshell> account.setName("Amanda")

jshell>
```

The method `setName` has the return type `void`, so it does not return a value, and no additional output appears in JShell.

The following gets the account object's name:

```
jshell> account.getName()
$6 ==> "Amanda"

jshell>
```

Method `getName` returns a `String`. When you invoke a method that returns a value, JShell stores the value in an implicitly declared variable. In this case, `$6`'s type is inferred to be `String`. Of course, you could have assigned the result of the preceding method call to an explicitly declared variable.

Using the Return Value of a Method in a Statement

If you invoke a method as part of a larger statement, the return value is used in that statement, rather than stored. For example, the following uses `println` to display the `account` object's name:

Click here to view code image

```
jshell> System.out.println(account.getName())
Amanda

jshell>
```

A.4.5 Creating a Meaningful Variable Name for an Expression

You can give a meaningful variable name to a value JShell previously assigned to an implicit variable. For example, with the following snippet recalled

```
jshell> account.getName()
```

Enter

Shift + *Tab*, *v*

This notation means you should press (or select) both the *Shift* and *Tab* keys together, then release those keys and press (or select) *v*. JShell infers the expression's type and begins a variable declaration for you—`account.getName()` returns a `String`, so JShell inserts `String` and an equal sign (=) before the expression, as in

Click here to view code image

```
jshell> account.getName()
jshell> String _= account.getName()
```

JShell also positions the cursor (indicated by the _ above) immediately before the = so you can simply enter the variable name, as in

Click here to view code image

```
jshell> String name = account.getName()
name ==> "Amanda"

jshell>
```

When you press (or select) *Enter*, JShell evaluates the new snippet and stores the value in the specified variable.

A.4.6 Saving and Opening Code-Snippet Files

You can save a session's valid code snippets to a file, which you can load into a JShell session as needed.

Saving Snippets to a File

To save just the valid snippets, use the **`/save` command**, as in:

```
/save filename
```

By default, the file is created in the folder from which you launched JShell. To store the file in a different location, specify the complete path of the file.

Loading Snippets from a File

Once you save your snippets, they can be reloaded with the **`/open` command**:

```
/open filename
```

which executes each snippet in the file.

Using `/open` to Load Java Source-Code Files

You can also open existing Java source code files using `/open`. For example, let's assume you'd like to experiment with class `Account` from Fig. 8.1 (as you did in Section A.4.1). Rather than entering its code into JShell, you can save time by loading the class from the source file `Account.java`. In a command window, you'd change to the folder containing `Account.java`, execute JShell, then use the following command to load the class declaration into JShell:

```
/open Account.java
```

To load a file from another folder, you can specify the file's full pathname. Section A.9 demonstrates how to use existing compiled classes in JShell.

A.5 Discovery with JShell Auto-Completion

[This section may be read after studying Section 2.8.]

JShell can help you write code. When you partially enter the name of an existing class, variable or method then press (or select) the *Tab* key, JShell does one of the following:

- If no other name matches what you've entered so far, JShell

enters the rest of the name for you.

- If there are multiple names that begin with the same letters, JShell displays a list of those names to help you decide what to enter next—then you can enter the next letter(s) and press (or select) *Tab* again to complete the name.
- If no names match what you entered so far, JShell does nothing, and your operating system's alert sound plays as feedback.

Auto-completion is typically an IDE feature, but with JShell, it's IDE-independent.

A.5.1 Auto-Completing Identifiers

Start a new JShell session or reset your current one. Then execute the following snippet to define the `String` variable `greeting` and initialize it to `"hello"`:

Click here to view code image

```
jshell> String greeting = "hello"
```

The only variable declared so far that begins with lowercase `"g"` is `greeting`—auto-completion is case-sensitive. If you enter `"g"` at the `jshell>` prompt:

```
jshell> g
```

then press (or select) *Tab*, JShell auto-completes the name:

```
jshell> greeting
```

If you then enter a dot:

```
jshell> greeting.
```

then press (or select) *Tab*, JShell does not know what method you want to call, so it displays a list of everything—in this case, all the methods—that can appear to the right of the dot:

Click here to view code image

```
jshell> greeting.
charAt(                 chars()                 codePointAt(
codePointBefore(        codePointCount(         codePoints()
```

```
compareTo(              compareToIgnoreCase(    concat(
contains(               contentEquals(          describeConstable()
endsWith(               equals(                 equalsIgnoreCase(
formatted(              getBytes(               getChars(
getClass()              hashCode()              indent(
indexOf(                intern()                isBlank()
isEmpty()               lastIndexOf(            length()
lines()                 matches(                notify()
notifyAll()             offsetByCodePoints(     regionMatches(
repeat(                 replace(                replaceAll(
replaceFirst(           resolveConstantDesc(    split(
splitWithDelimiters(    startsWith(             strip()
stripIndent()           stripLeading()          stripTrailing()
subSequence(            substring(              toCharArray()
toLowerCase(            toString()              toUpperCase(
transform(              translateEscapes()      trim()
wait(

jshell> greeting.
```

and follows the list with a new jshell> prompt that includes what you've entered so far. The list includes the methods declared in class String and several Object methods that all Java classes have (discussed in Chapter 9). In the list of method names

- those followed by "()" are methods that do not require arguments and
- those followed only by "(" are methods that either require at least one argument or are overloaded methods that can be called with different numbers and/or types of arguments (discussed in Section 5.7).

Let's assume you want to use String's endsWith method to check whether greeting starts with "o". There's only one method that begins with "en", so you can enter en then *Tab* to auto-complete endsWith:

```
jshell> greeting.endsWith(
```

JShell automatically inserts the method call's opening left parenthesis. Now you can complete the snippet as in:

Click here to view code image

```
jshell> greeting.endsWith("o")
$2 ==> true
```

```
jshell>
```

A.5.2 Auto-Completing JShell Commands

Auto-completion also works for JShell commands. If you enter `/` then press (or select) *Tab*, JShell displays the list of JShell commands:

Click here to view code image

```
jshell> /
/!          /?          /drop       /edit      /env        /exit
/help       /history    /imports    /list      /methods    /open
/reload     /reset      /save       /set       /types      /vars

<press tab again to see synopsis>
jshell> /
```

If you then enter `h` and press (or select) *Tab*, JShell displays only the commands that start with `/h`:

Click here to view code image

```
jshell> /h
/help       /history

<press tab again to see synopsis>
jshell> /h
```

Finally, if you enter `"i"` and press (or select) *Tab*, JShell auto-completes `/history`. Similarly, if you enter `/l` then press (or select) *Tab*, JShell auto-completes the command as `/list` because only that command starts with `/l`.

A.6 Exploring a Class's Members and Viewing Documentation

[This section may be read after studying Chapter 5, Methods.]

The preceding section introduced basic auto-completion capabilities. When using JShell for experimentation and discovery, you'll often want to learn more about a class before using it. In this section, we'll explore class `Math` to demonstrate how to view:

- the parameters required by a method so that you can call it correctly
- the documentation for a method
- the documentation for a field of a class
- the documentation for a class, and
- the list of overloads for a given method.

Start a new JShell session or `/reset` the current one.

A.6.1 Listing Class `Math`'s `static` Members

As discussed in Chapter 5, class `Math` contains only `static` members—`static` methods for various mathematical calculations and the `static` constants `PI`, `E` and `TAU`. To view a complete list, start a new JShell session, then enter `"Math."` and press (or select) *Tab*:

Click here to view code image

```
jshell> Math.
<press tab again to see all possible completions; total possible
completions: 103>
jshell> Math.
```

Because there are so many possible completions, JShell asks you to press *Tab* again to see all possible completions. Once you do, it displays—only 62 of the 103 completions are displayed because many of class `Math`'s methods are overloaded:

Click here to view code image

```
jshell> Math.
E                   IEEEremainder(      PI                  TAU
abs(                absExact(           acos(               addExact(
asin(               atan(               atan2(              cbrt(
ceil(               ceilDiv(            ceilDivExact(       ceilMod(
clamp(              class               copySign(           cos(
cosh(               decrementExact(     divideExact(        exp(
expm1(              floor(              floorDiv(           floorDivExact(
floorMod(           fma(                getExponent(        hypot(
incrementExact(     log(                log10(              log1p(
max(                min(                multiplyExact(      multiplyFull(
multiplyHigh(       negateExact(        nextAfter(          nextDown(
nextUp(             pow(                random()            rint(
round(              scalb(              signum(             sin(
```

```
sinh(              sqrt(              subtractExact(    tan(
tanh(              toDegrees(         toIntExact(       toRadians(
ulp(               unsignedMultiplyHigh(
jshell> Math.
```

Again, JShell auto-completion displays everything that can appear to the right of the dot (`.`). Here, we entered a class name and a dot (`.`), so JShell displays only the class's `static` members. The names that are not followed by any parentheses (`E`, `PI` and `TAU`) are the class's `static` variables. All the other names are the class's `static` methods:

- Any method names followed by `()`—only `random` in this case—do not require arguments.
- Any method names followed by only an opening left parenthesis, `(`, require at least one argument or are overloaded.

You can easily view the value of constants like `PI` and `E`:

```
jshell> Math.PI
$1 ==> 3.141592653589793

jshell> Math.E
$2 ==> 2.718281828459045

jshell>
```

A.6.2 Viewing a Method's Parameters

Let's assume you wish to test `Math`'s `pow` method but do not know the required parameters.
You can enter

```
Math.p
```

then press (or select) *Tab* to auto-complete the name `pow`:

```
jshell> Math.pow(
```

No other methods begin with `"pow"`, so JShell inserts the left parenthesis to indicate the beginning of a method call. Next, you can press (or select) *Tab* to view the method's parameters:

Click here to view code image

```
jshell> Math.pow(
$1   $2

Signatures:
double Math.pow(double a, double b)

<press tab again to see documentation>
jshell> Math.pow(
```

JShell displays

- variables in the current session (`$1` and `$2`) that could be used as `pow`'s arguments,
- the method's return type, name and complete parameter list, indicating that the method requires two `double` parameters,
- `"<press tab again to see documentation>"`, and
- the next `jshell>` prompt containing what you've entered so far.

A.6.3 Viewing a Method's Documentation

JShell integrates the Java API documentation so you can view documentation conveniently in JShell rather than using a separate web browser. Suppose you'd like to learn more about `pow` before completing your code snippet. You can press (or select) *Tab* again to view the method's Java documentation (known as its javadoc). We cut out some of the documentation text and replaced it with an ellipsis (…) to save space—try the steps in your own JShell session to observe the complete text:

Click here to view code image

```
jshell> Math.pow(
double Math.pow(double a, double b)
Returns the value of the first argument raised to the power of the
second argument. Special cases:
  * If the second argument is positive or negative zero, then the result
is 1.0.
...
<press tab again to see next page>
jshell> Math.pow(
```

For lengthy documentation, JShell displays part of it followed by the message:

Click here to view code image

```
<press tab again to see next page>
```

You can press (or select) *Tab* to view the next page of documentation. The next `jshell>` prompt contains the portion of the snippet you've entered so far.

A.6.4 Viewing a `public` Field's Documentation

You can use the *Tab* feature to learn more about a class's `public` fields. For example, if you enter `Math.PI` followed by *Tab*, JShell displays

Click here to view code image

```
jshell> Math.PI
Signatures:
Math.PI:double

<press tab again to see documentation>
```

which includes `Math.PI`'s type and indicates that you can use *Tab* again to view the documentation. Doing so displays:

Click here to view code image

```
jshell> Math.PI
Math.PI:double
The double value that is closer than any other to pi (π), the ratio of
the circumference of a circle to its diameter.
jshell> Math.PI
```

and the next `jshell>` prompt contains the portion of the snippet you've entered so far.

A.6.5 Viewing a Class's Documentation

You can also enter a class name then *Tab* to view the class's fully qualified name. For example, entering `Math` then *Tab* outputs:

Click here to view code image

```
jshell> Math
```

```
Math          MathContext

Signatures:
java.lang.Math

<press tab again to see documentation>
jshell> Math
```

indicating that class `Math` is in the package `java.lang`. Entering *Tab* again contains the beginning of the class's documentation:

Click here to view code image

```
jshell> Math
java.lang.Math
The class Math contains methods for performing basic numeric operations
such as the elementary exponential, logarithm, square root, and
trigonometric functions.
...

<press tab again to see next page>
jshell> Math
```

In this case, there is more documentation to view, so you can press (or select) *Tab* to view it. Whether or not you view the remaining documentation, the `jshell>` prompt contains the portion of the snippet you've entered so far.

A.6.6 Viewing Method Overloads

Many classes have overloaded methods. When you press (or select) *Tab* to view an overloaded method's parameters, JShell displays the complete list of overloads with the parameters for every overload. For example, method `Math.abs` has four overloads:

Click here to view code image

```
jshell> Math.abs(
$1   $2

Signatures:
int Math.abs(int a)
long Math.abs(long a)
float Math.abs(float a)
double Math.abs(double a)

<press tab again to see documentation>
```

```
jshell> Math.abs(
```

When you press (or select) *Tab* again to view the documentation, JShell outputs the first overload's documentation:

Click here to view code image

```
jshell> Math.abs(
int Math.abs(int a)
Returns the absolute value of an int value.If the argument is not
negative, the argument is returned. If the argument is negative, the
negation of the argument is returned.
...

<press tab again to see next page>
```

You can then press (or select) *Tab* to view the documentation for the next overload in the list. Again, whether or not you view the remaining documentation, the `jshell>` prompt contains the portion of the snippet you've entered so far.

A.6.7 Exploring Members of a Specific Object

The exploration features in Sections A.6.1–A.6.6 also apply to the members of a specific object. Let's start a new session then create and explore a `String` object:

Click here to view code image

```
jshell> String dayName = "Monday"
dayName ==> "Monday"

jshell>
```

To view the `dayName` object's methods, enter `"dayName."` and press (or select) *Tab*:

Click here to view code image

```
jshell> dayName.
charAt(                 chars()                 codePointAt(
codePointBefore(        codePointCount(         codePoints()
compareTo(              compareToIgnoreCase(    concat(
contains(               contentEquals(          describeConstable()
endsWith(               equals(                 equalsIgnoreCase(
formatted(              getBytes(               getChars(
getClass()              hashCode()              indent(
indexOf(                intern()                isBlank()
```

```
isEmpty()                  lastIndexOf(              length()
lines()                    matches(                  notify()
notifyAll()                offsetByCodePoints(       regionMatches(
repeat(                    replace(                  replaceAll(
replaceFirst(              resolveConstantDesc(      split(
splitWithDelimiters(       startsWith(               strip()
stripIndent()              stripLeading()            stripTrailing()
subSequence(               substring(                toCharArray()
toLowerCase(               toString()                toUpperCase(
transform(                 translateEscapes()        trim()
wait(

jshell> dayName.
```

Exploring `toUpperCase`

Let's investigate the `toUpperCase` method. Continue by entering `"toU"` and pressing *Tab* to auto-complete its name:

```
jshell> dayName.toUpperCase(
```

Then, enter *Tab* to view the method's parameters:

Click here to view code image

```
jshell> dayName.toUpperCase(
Signatures:
String String.toUpperCase(Locale locale)
String String.toUpperCase()

<press tab again to see documentation>
jshell> dayName.toUpperCase(
```

This method has two overloads. You can now use *Tab* to read about each overload or choose the one you wish to use by specifying the appropriate arguments (if any). In this case, we'll use the no-argument version to create a new `String` containing `MONDAY`, so we simply enter the closing right parenthesis of the method call and press (or select) *Enter*:

```
jshell> dayName.toUpperCase()
$2 ==> "MONDAY"

jshell>
```

Exploring `substring`

Let's assume you want to create the new `String "DAY"`—a subset of the implicit variable `$2`'s characters. For this purpose, class `String` provides the overloaded method `substring`. First, enter `"$2.subs"` and press (or select) *Tab* to auto-complete the method's name:

```
jshell> $2.substring(
```

Next, use *Tab* to view the method's overloads:

Click here to view code image

```
jshell> $2.substring(
Signatures:
String String.substring(int beginIndex)
String String.substring(int beginIndex, int endIndex)

<press tab again to see documentation>
jshell> $2.substring(
```

Next, use *Tab* again to view the first overload's documentation:

Click here to view code image

```
jshell> $2.substring(
String String.substring(int beginIndex)
Returns a string that is a substring of this string.The substring begins
with the character at the specified index and extends to the end of this
string.
...
<press tab again to see next page>
```

The documentation indicates that this overload of the method enables you to obtain a substring starting from a specific character index (that is, position) and continuing through the end of the `String`. The first character in the `String` is at index `0`. This is the version of the method we wish to use to obtain `"DAY"` from `"MONDAY"`, so we can return to our code snippet at the `jshell>` prompt:

```
jshell> $2.substring(
```

Finally, we can complete our call to substring and press (or select) *Enter* to view the results:

```
jshell> $2.substring(3)
$3 ==> "DAY"
```

```
jshell>
```

A.7 Declaring Methods

[This section may be read after studying Chapter 5, Methods, and the preceding portions of this appendix.]

You can use JShell to prototype methods. For example, let's assume we'd like to write code that displays the cubes of the values from 1 through 10. For the purpose of this discussion, we're going to define two methods:

- Method `displayCubes` will iterate 10 times, calling method `cube` each time.
- Method `cube` will receive one int value and return the cube of that value.

A.7.1 Forward Referencing an Undeclared Method—Declaring Method `displayCubes`

Let's begin with method `displayCubes`. Start a new JShell session or `/reset` the current one, then enter the following method declaration:

Click here to view code image

```
void displayCubes() {
   for (int i = 1; i <= 10; ++i) {
      System.out.printf("Cube of %d is %d%n", i, cube(i));
   }
}
```

When you complete the method declaration, JShell displays:

Click here to view code image

```
|  created method displayCubes(), however, it cannot be invoked until
method cube(int) is declared

jshell>
```

After you enter the method body's opening left brace, JShell displays continuation prompts (`...>`) before each subsequent line until you complete the method declaration by entering its closing

right brace. Also, although JShell says `"created method displayCubes()"`, it indicates that you cannot call this method until `"cube(int) is declared"`. This is not fatal in JShell—it recognizes that `displayCubes` depends on an undeclared method (`cube`)—this is known as **forward referencing** an undeclared method. Once you define `cube`, you can call `displayCubes`.

A.7.2 Declaring a Previously Undeclared Method

Next, let's declare method `cube` but purposely make a logic error by returning the square rather than the cube of its argument:

```
jshell> int cube(int x) {
   ...>    return x * x;
   ...> }
|  created method cube(int)

jshell>
```

At this point, you can use JShell's **`/methods` command** to list the methods declared in the current JShell session:

```
jshell> /methods
|    void displayCubes()
|    int cube(int)

jshell>
```

A.7.3 Testing `cube` and Replacing Its Declaration

Now that method `cube` is declared, let's test it with the argument `2`:

```
jshell> cube(2)
$3 ==> 4

jshell>
```

Based on how the method is currently implemented, it returns the `4` (that is, 2 * 2). However, the method's purpose was to calculate the cube of its argument, so the result should have been `8` (2 * 2 * 2). You can edit `cube`'s snippet to correct the problem. Let's edit the snippet directly in JShell rather than in JShell Edit Pad. Use the up

arrow three times to redisplay cube's definition:

```
jshell> int cube(int x) {
   ...>    return x * x;
   ...> }
```

Then use the arrow keys to move the cursor to the semicolon and edit the statement so the method appears as follows, then press (or select) *Enter*:

```
jshell> int cube(int x) {
   ...>   return x * x * x;
   ...> }
|  modified method cube(int)

jshell>
```

JShell confirms that method `cube` was modified.

A.7.4 Testing Updated Method `cube` and Method `displayCubes`

Now that method `cube` is properly declared, let's test it again with the arguments `2` and `10`:

```
jshell> cube(2)
$5 ==> 8

jshell> cube(10)
$6 ==> 1000

jshell>
```

The method properly returns the cubes of `2` (that is, `8`) and `10` (that is, `1000`) and stores the results in the implicit variables `$5` and `$6`.

Now, let's test `displayCubes`. If you enter `"di"` and press (or select) *Tab*, JShell auto-completes the name, including the parentheses of the method call, because `displayCubes` receives no parameters. The following contains results of the call:

```
jshell> displayCubes()
Cube of 1 is 1
Cube of 2 is 8
Cube of 3 is 27
Cube of 4 is 64
```

```
Cube of 5 is 125
Cube of 6 is 216
Cube of 7 is 343
Cube of 8 is 512
Cube of 9 is 729
Cube of 10 is 1000

jshell>
```

A.8 Exceptions

[This section may be read after studying Chapter 6 and the preceding sections of this appendix.]

In Section 6.11, we introduced Java's exception-handling mechanism, demonstrating how to catch an exception that occurred when we attempted to use an out-of-bounds array index. In JShell, catching exceptions is not required—it automatically catches each exception and displays information about it, then displays the next JShell prompt, so you can continue your session. This is particularly important for checked exceptions (Section 10.5) that must be caught in regular Java programs. As you know, catching an exception requires wrapping the code in a `try`...`catch` statement. By automatically, catching all exceptions, JShell makes it easier for you to experiment with methods that throw checked exceptions.

In the following new JShell session, we declare an array of `int` values then demonstrate both valid and invalid array-access expressions:

Click here to view code image

```
jshell> int[] values = {10, 20, 30}
values ==> int[3] { 10, 20, 30 }

jshell> values[1]
$2 ==> 20

jshell> values[10]
|  Exception java.lang.ArrayIndexOutOfBoundsException: Index 10 out of
bounds for length 3
|        at (#3:1)

jshell>
```

The snippet `values[10]` attempts to access an out-of-bounds element—recall that this results in an

`ArrayIndexOutOfBoundsException`. Even though we did not wrap the code in a `try`…`catch`, JShell catches the exception and displays its `String` representation. This includes the exception's type and an error message (in this case, the invalid index `10`), followed by a stack trace indicating where the problem occurred. The notation

```
|      at (#3:1)
```

indicates that the exception occurred at line 1 of the code snippet with the ID 3. Section 5.10 discussed the method-call stack. A stack trace indicates the methods that were on the method-call stack at the time the exception occurred. A typical stack trace contains several `"at"` lines like the one output here—one per stack frame. After displaying the stack trace, the next `jshell>` prompt appears. Chapter 10 discusses stack traces in detail.

A.9 Importing Classes and Adding Packages to the `CLASSPATH`

[This section may be read after studying Chapter 11, Files, I/O Streams, JSON Serialization & CSV Files, and the preceding sections of this appendix.]

JShell is helpful for experimenting with classes and class libraries before using them in your programs. You can `import` types from Java's API and third-party libraries that you add to JShell's classpath and module path. Several packages are so commonly used by Java developers that JShell automatically `import`s them for you. You can change this with JShell's

```
/set start
```

command, which we present in Section A.11.

Viewing the Imported Packages

For this section, we assume you started `jshell` from the `ch11` examples folder. The JShell **`/imports` command** lists the current session's `import` declarations. The following packages are auto-imported when you begin a new JShell session:

Click here to view code image

```
jshell> /imports
|    import java.io.*
|    import java.math.*
|    import java.net.*
|    import java.nio.file.*
|    import java.util.*
|    import java.util.concurrent.*
|    import java.util.function.*
|    import java.util.prefs.*
|    import java.util.regex.*
|    import java.util.stream.*

jshell>
```

The `java.lang` package's contents are always available in JShell, just as in any Java source-code file.

Adding Packages to the Class Path

You can also import your own or third-party packages to experiment with them in JShell. JShell's **`/env -class-path` command** adds packages to JShell's `CLASSPATH`, which tells JShell where additional packages are located.[7] You can then use `import` declarations to experiment with the packages' contents in JShell.

7. JShell also provides the options **`-module-path`** and **`-add-modules`** to add modules so you can import packages from them.

Using Jackson in JShell

In Chapter 11, we introduced the Jackson open-source library for serializing objects to JSON format. Here, we'll add it to JShell's `CLASSPATH`, then serialize an object to a JSON `String` in JShell. If you have an open JShell session, use `/exit` to terminate it. Then, change directories to the `ch11` examples folder and start a new JShell session.

Adding the Location of a Package to the `CLASSPATH`

The `ch11` folder's `libs` subfolder contains the Jackson library components we used in Chapter 11. The following command adds this package to the `CLASSPATH` on Windows—ensure that you type the entire classpath without spaces:

Click here to view code image

```
jshell> /env -class-path libs/jackson-core-2.18.1.jar;libs/jackson-
databind-2.18.1.jar;libs/jackson-annotations-2.18.1.jar
|  Setting new options and restoring state.

jshell>
```

If you use macOS, replace the semicolons (`;`) with colons (`:`). The `.jar` (Java archive) files are ZIP files containing compiled Jackson library Java code. We use relative paths to the `libs` folder here. You can also specify absolute paths.

Importing a Class from the Package

Now, you can import `ObjectMapper` for use in JShell. The following demonstrates importing the `ObjectMapper` class and the complete list of imports in the current session:

Click here to view code image

```
jshell> import com.fasterxml.jackson.databind.ObjectMapper;
jshell> /imports
|    import java.io.*
|    import java.math.*
|    import java.net.*
|    import java.nio.file.*
|    import java.util.*
|    import java.util.concurrent.*
|    import java.util.function.*
|    import java.util.prefs.*
|    import java.util.regex.*
|    import java.util.stream.*
|    import com.fasterxml.jackson.databind.ObjectMapper

jshell>
```

Creating a `List<Account>` to Serialize as JSON

Let's create a `List<Account>` to serialize in JShell:

Click here to view code image

```
jshell> record Account(int accountNumber, String name, double balance) {}
|  created record Account

jshell> List<Account> accounts = List.of(
   ...>    new Account(100, "Devi", 24.98),
   ...>    new Account(200, "Taylor", 345.67));
accounts ==> [Account[accountNumber=100, name=Devi, balance=24 ...
e=Taylor, balance=345.67]]
```

```
jshell>
```

Using the Imported `ObjectMapper` Class

Finally, you can use class `ObjectMapper`. Below, we create an `ObjectMapper` and demonstrate that JShell's auto-complete capability can display the list of available methods. There are 247 autocomplete options, so we show just a few below and use an ellipsis to indicate there are more:

Click here to view code image

```
jshell> var mapper = new ObjectMapper();
mapper ==> com.fasterxml.jackson.databind.ObjectMapper@7e5afaa6

jshell> mapper.
<press tab again to see all possible completions; total possible
completions: 247>
jshell> mapper.
acceptJsonFormatVisitor(             activateDefaultTyping(
activateDefaultTypingAsProperty(     addHandler(
addMixIn(                            addMixInAnnotations(
...
jshell> mapper.
```

Next, we use **`ObjectMapper`'s `writerWithDefaultPrettyPrinter` method** to get an `ObjectWriter` that outputs JSON text with indentation for better readability:

Click here to view code image

```
jshell> var prettyWriter = mapper.writerWithDefaultPrettyPrinter()
prettyWriter ==> com.fasterxml.jackson.databind.ObjectWriter@57d7f8ca

jshell>
```

Finally, we use the **`ObjectWriter`'s `writeValueAsString` method** to get `accounts`' JSON representation and display the returned `String`:

Click here to view code image

```
jshell> System.out.println(prettyWriter.writeValueAsString(accounts))
[ {
  "accountNumber" : 100,
  "name" : "Devi",
  "balance" : 24.98
}, {
```

```
  "accountNumber" : 200,
  "name" : "Taylor",
  "balance" : 345.67
} ]

jshell>
```

A.10 Using an External Editor

Section A.2.10 demonstrated JShell Edit Pad for editing code snippets. This tool provides only simple editing functionality. Many programmers prefer to use more powerful text editors. You can specify your preferred text editor using JShell's **`/set editor` command**. For example, the JShell command

Click here to view code image

```
jshell> /set editor notepad.exe
|  Editor set to: notepad.exe

jshell>
```

sets `notepad.exe` as the snippet editor for the current JShell session. The `/set editor` command's argument is operating-system-specific. For example, on macOS, you can use the built-in `nano` text editor with the command

```
/set editor nano
```

Editing Snippets with a Custom Text Editor

When you're using a custom editor, each time you save snippet edits JShell immediately re-evaluates any snippets that have changed and outputs their results (but not the snippets themselves) in the JShell output.

Retaining the Editor Setting

You can retain your editor setting for future JShell sessions as follows:

Click here to view code image

```
/set editor -retain commandToLaunchYourEditor
```

Restoring the JShell Edit Pad As the Default Editor

If you do not retain your custom editor, subsequent JShell sessions will use JShell Edit Pad. If you do retain the custom editor, you can restore JShell Edit Pad as the default with

```
/set editor -retain -default
```

A.11 Summary of JShell Commands

The following table contains the basic JShell commands. Many of these commands have been presented throughout this appendix. Others are discussed in this section.

Command	Description
`/help`	Displays JShell's list of commands.
`/help intro`	Displays a brief introduction to JShell.
`/help shortcuts`	Displays a description of several JShell shortcut keys.
`/list`	By default, lists the valid snippets you've entered in the current session. To list all snippets, use `/list -all`.
`/!`	Recalls and re-evaluates the last snippet.
`/id`	Recalls and re-evaluates the snippet with the specified *id*. You may also specify several individual snippets or snippet ranges, each separated from the next by a space.
`/-n`	Recalls and re-evaluates a prior snippet—for *n*, `1` is the last snippet, `2` is the second-to-last, etc.
`/edit`	By default, opens a JShell Edit Pad window containing the valid snippets you've entered in the current session. Section A.10 discusses how to configure an external editor.
`/save`	Saves the current session's valid snippets to a specified file.
`/open`	Opens a specified file of code snippets, loads the snippets into the current session and evaluates the loaded snippets.
`/vars`	Displays the current session's variables and their corresponding values.
`/methods`	Displays the signatures of the current session's declared methods.
`/types`	Displays types declared in the current session.
`/imports`	Displays the current session's `import` declarations.
`/exit`	Terminates the current JShell session.
`/reset`	Resets the current JShell session, deleting all code snippets.
`/reload`	Reloads a JShell session and executes the valid snippets (Section

A.11.3).

`/drop` Deletes a specified snippet from the current session (Section A.11.4).

`/env` Makes changes to the JShell environment, such as adding packages or modules so you can use their types in JShell.

`/list -all` Lists everything you've entered in the current JShell session, including all snippets (valid, invalid or overwritten) and JShell commands—the `/list` command outputs only snippets, not JShell commands.

`/set` Sets various JShell configuration options, such as the editor used in response to the `/edit` command, the text used for the JShell prompts, the imports to specify when a session starts, etc. (Sections A.11.5–A.11.6).

A.11.1 Getting Help in JShell

JShell's help documentation is incorporated directly via the **`/help`** or **`/?`** commands—`/?` is simply a shorthand for `/help`. For a quick introduction to JShell, enter:

```
/help intro
```

To display JShell's list of commands, enter

```
/help
```

For more information on a given command's options, enter

```
/help command
```

For example

```
/help /list
```

displays the `/list` command's more detailed help documentation. Similarly

```
/help /set start
```

displays more detailed help documentation for the `/set` command's `start` option. For a list of the shortcut key combinations in JShell, enter

```
/help shortcuts
```

A.11.2 `/edit` Command: Additional Features

We've discussed using `/edit` to load all valid snippets, a snippet with a specified ID or a method with a specified name into JShell Edit Pad. You can specify the identifier for any variable, method or type declaration you'd like to edit. For example, if the current JShell session contains the declaration of a class named `Account`, the following loads that class into JShell Edit Pad:

```
/edit Account
```

A.11.3 `/reload` Command

JShell's **`/reload`** command can re-execute all valid snippets in the current session. Consider the session from Sections A.2.9–A.2.10:

Click here to view code image

```
jshell> /list

   1 : 45
   2 : 72
   3 : if ($1 < $2) {
           System.out.printf("%d < %d%n", $1, $2);
       }
   4 : if ($1 > $2) {
           System.out.printf("%d > %d%n", $1, $2);
       }
   5 : $1 = 100;
   6 : if ($1 > $2) {
           System.out.printf("%d > %d%n", $1, $2);
       }

jshell>
```

The following reloads that session one snippet at a time:

Click here to view code image

```
jshell> /reload
|  Restarting and restoring state.
-: 45
-: 72
-: if ($1 < $2) {
      System.out.printf("%d < %d%n", $1, $2);
   }
45 < 72
-: if ($1 > $2) {
```

```
        System.out.printf("%d > %d%n", $1, $2);
    }
-: $1 = 100
-: if ($1 > $2) {
        System.out.printf("%d > %d%n", $1, $2);
    }
100 > 72

jshell>
```

Each reloaded snippet is preceded by `-:` and in the case of the `if` statements, the output (if any) appears immediately following each `if` statement. If you prefer not to list the snippets as they reload, you can use the `/reload` command's `-quiet` option:

Click here to view code image

```
jshell> /reload -quiet
|  Restarting and restoring state.
45 < 72
100 > 72

jshell>
```

In this case, only the results of output statements are displayed. Then, you can view the snippets that were reloaded with the `/list` command.

A.11.4 `/drop` Command

You can eliminate a snippet from the current session with JShell's `/drop` command followed by a snippet ID or an identifier. The following new JShell session declares a variable `x` and a method `cube`, then drops `x` via its snippet ID and drops `cube` via its identifier:

Click here to view code image

```
jshell> int x = 10
x ==> 10

jshell> int cube(int y) {return y * y * y;}
|  created method cube(int)

jshell> /list

   1 : int x = 10;
   2 : int cube(int y) {return y * y * y;}
```

```
jshell> /drop 1
|  dropped variable x

jshell> /drop cube
|  dropped method cube(int)

jshell> /list

jshell>
```

A.11.5 Feedback Modes

JShell has several feedback modes that determine what gets displayed after each interaction. To change the feedback mode, use JShell's `/set feedback` command:

```
/set feedback mode
```

where *mode* is `concise`, `normal` (the default), `silent` or `verbose`. All of the prior JShell interactions in this appendix used the `normal` mode.

Feedback Mode `verbose`

Below is a new JShell session in which we used `verbose` mode, which developers new to Java might prefer:

Click here to view code image

```
jshell> /set feedback verbose
|  Feedback mode: verbose

jshell> int x = 10
x ==> 10
|  created variable x : int

jshell> int cube(int y) {return y * y * y;}
|  created method cube(int)

jshell> cube(x)
$3 ==> 1000
|  created scratch variable $3 : int

jshell> x = 5
x ==> 5
|  assigned to x : int
```

```
jshell> cube(x)
$5 ==> 125
|  created scratch variable $5 : int

jshell>
```

Notice the additional feedback indicating that

- variable `x` was created,
- method `cube` was created,
- scratch variable `$3` was created on the first call to `cube`,
- an `int` was assigned to the variable `x`, and
- scratch variable `$5` was created on the second call to `cube`.

Feedback Mode `concise`

Next, we `/reset` the session then set the feedback mode to `concise` and repeat the preceding session:

Click here to view code image

```
jshell> /set feedback concise
jshell> int x = 10
jshell> int cube(int y) {return y * y * y;}
jshell> cube(x)
$3 ==> 1000
jshell> x = 5
jshell> cube(x)
$5 ==> 125
jshell>
```

Note that the only feedback displayed is the result of each call to `cube`. If an error occurs, its feedback also will be displayed.

Feedback Mode `silent`

Finally, we `/reset` the session then set the feedback mode to `silent` and repeat the preceding session:

Click here to view code image

```
jshell> /set feedback silent
-> int x = 10
-> int cube(int y) {return y * y * y;}
-> cube(x)
-> x = 5
```

```
-> cube(x)
-> /set feedback normal
|  Feedback mode: normal

jshell>
```

In this case, the `jshell>` prompt becomes `->` and only error feedback will be displayed. You might use this mode if you've copied code from a Java source file and want to paste it into JShell, but do not want feedback for each line.

A.11.6 Other JShell Features Configurable with `/set`

So far, we've demonstrated the `/set` command's capabilities for setting an external snippet editor and setting feedback modes. The `/set` command provides extensive capabilities for creating custom feedback modes via the commands:

- `/set mode`
- `/set prompt`
- `/set truncation`
- `/set format`

The `/set mode` command creates a user-defined custom feedback mode. Then, you can use the other three commands to customize all aspects of JShell's feedback. The details of these commands are beyond the scope of this appendix. JShell's help documentation for each of the preceding commands contains more information.

Customizing JShell Startup

Section A.9 listed the set of common packages JShell `import`s at the start of each session. Using JShell's **`/set start`** command

```
/set start filename
```

you can provide a file of Java snippets and JShell commands that will be used in the current session when it restarts due to a `/reset` or `/reload` command. You can also remove all startup snippets with

```
/set start -none
```

or return to the default startup snippets with

```
/set start -default
```

In all three cases, the setting applies only to the current session unless you also include the `-retain` option. For example, the following command indicates that all subsequent JShell sessions should load the specified file of startup snippets and commands:

```
/set start -retain filename
```

You can restore the defaults for future sessions with

```
/set start -retain -default
```

A.12 Keyboard Shortcuts for Snippet Editing

In addition to the commands in Fig. A.11, JShell supports many keyboard shortcuts for editing code, such as quickly jumping to the beginning or end of a line or jumping between words in a line. The following table contains some of the shortcuts available.

Shortcut	Description
Ctrl + *a*	Move cursor to beginning of line.
Ctrl + *e*	Move cursor to end of line.
Alt + *b* (Windows) or *Option* + ← (macOS)	Move the cursor backward by one word.
Alt + *f* (Windows) or *Option* + → (macOS)	Move the cursor forward by one word.
Ctrl + *r*	Perform a search for the last command or snippet containing the characters you enter after *Ctrl* + *r*.
Ctrl + *t*	Reverse the two characters to the left of the cursor.
Ctrl + *k*	Cut everything from the cursor to the end of the line.
Ctrl + *u*	Cut everything from the beginning of the line up to, but not including the character at the cursor position.
Ctrl + *w*	Cut the word before the cursor.
Alt + *d* (Windows) or *Option* + *d* (macOS)	Cut the word after the cursor.

A.13 How JShell Reinterprets Java for Interactive Use

In JShell:

- A `main` method is not required.
- Semicolons are not required on standalone statements.
- Variables do not need to be declared in classes or in methods.
- Methods do not need to be declared inside a class's body.
- Statements do not need to be written inside methods.
- Redeclaring a variable, method or type drops the prior declaration and replaces it with the new one, whereas the Java compiler typically would report an error.
- You do not need to catch exceptions, though you can if you need to test exception handling.
- JShell ignores top-level access modifiers (`public`, `private`, `protected`, `static`, `final`)—only `abstract` (Chapter 9) is allowed as a class modifier.
- The `synchronized` keyword (Chapter 18, Concurrency: Platform Threads to Virtual Threads) is ignored.
- `package` statements and `module` statements are not allowed.

A.14 IDE JShell Support

Many Java IDEs include JShell support, including IntelliJ IDEA, the Eclipse IDE for Java Developers, NetBeans and Visual Studio Code. Each handles opening JShell and running code snippets in its own way. Some include code syntax coloring and handle name auto-completion differently.

Generative AI

1 Prompt genAIs for a step-by-step tutorial on using JShell in your preferred IDE. Ask them to explain the key differences between JShell in your IDE vs. JShell at the command line.

A.15 Wrap-Up

In this appendix, you used JShell—Java's interactive REPL for exploration, discovery and experimentation. We started a JShell session and worked with various types of code snippets, including statements, variables, expressions, methods and classes—all without having to declare a class containing a `main` method to execute the

code.

You listed the valid snippets in the current session, and recalled and executed prior snippets and commands using the up and down arrow keys. You also learned that you can list the current session's variables, methods, types and `import`s. We cleared the current JShell session to remove all existing snippets and saved snippets to a file then reloaded them.

We demonstrated JShell's auto-completion capabilities for code and commands, explored a class's members and displayed documentation directly in JShell. We explored class `Math`, demonstrating how to list its `static` members, how to view a method's parameters and overloads, view a method's documentation and view a `public` field's documentation. We also explored the methods of a `String` object.

You declared methods and forward referenced an undeclared method that you declared later in the session, then learned that you could go back and execute the first method. We also replaced a method declaration with a new method—in fact, you can replace any declaration of a variable, method or type.

We demonstrated that JShell catches all exceptions and simply displays a stack trace followed by the next `jshell>` prompt, so you can continue the session. You imported an existing compiled class from a package, then used that class in a JShell session.

Next, we summarized and demonstrated various other JShell commands. We configured a custom snippet editor, viewed JShell's help documentation, reloaded a session, dropped snippets from a session, configured feedback modes and more. We listed some additional keyboard shortcuts for editing the current snippet at the `jshell>` prompt. Finally, we discussed how JShell reinterprets Java for interactive use and mentioned IDE support for JShell.

Self-Review Exercises with Answers

We encourage you to use JShell to do Exercises A.1–A.43 after reading Sections A.2–A.3. We've included the answers for all these exercises to help you get comfortable with JShell quickly.

A.1 Confirm that when you use `System.out.println` to display a `String` literal, such as `"Happy Birthday!"`, the quotes (`""`) are not displayed. End your statement with a semicolon.

Answer:

Click here to view code image

```
jshell> System.out.println("Happy Birthday!");
Happy Birthday!

jshell>
```

A.2 Repeat Exercise A.1, but remove the semicolon at the end of your statement to demonstrate that semicolons in this position are optional in JShell.

Answer:

Click here to view code image

```
jshell> System.out.println("Happy Birthday!")
Happy Birthday!

jshell>
```

A.3 Confirm that JShell does not execute a `//` end-of-line comment.

Answer:

Click here to view code image

```
jshell> // comments are not executable

jshell>
```

A.4 Show that an executable statement enclosed in a multiline comment—delimited by `/*` and `*/`—does not execute.

Answer:

Click here to view code image

```
jshell> /* opening line of multi-line comment
   ...>    System.out.println("Welcome to Java Programming!")
   ...>    closing line of multi-line comment */

jshell>
```

A.5 Show what happens when the following code is entered in JShell:

Click here to view code image

```
/* incomplete multi-line comment
System.out.println(“Welcome to Java Programming!”)
/* complete multi-line
```

```
comment */
```

Answer: There is no compilation error because the second `/*` is considered to be part of the first multi-line comment.

Click here to view code image

```
jshell> /* incomplete multi-line comment
   ...> System.out.println("Welcome to Java Programming!")
   ...> /* complete multi-line
   ...> comment */

jshell>
```

A.6 Show that indenting code with spaces does not affect statement execution.

Answer:

Click here to view code image

```
jshell> System.out.println("A")
A

jshell>   System.out.println("A") // indented 3 spaces
A

jshell>      System.out.println("A") // indented 6 spaces
A

jshell>
```

A.7 Declare each of the following variables as type `int` in JShell to determine which are valid and which are invalid?

a. `first`
b. `first number`
c. `first1`
d. `1first`

Answer: a) valid. b) invalid (space not allowed). c) valid. d) invalid (can't begin with a digit).

```
jshell> int first
first ==> 0

jshell> int first number
|  Error:
|  ';' expected
```

```
|  int first number
|            ^

jshell> int first1
first1 ==> 0

jshell> int 1first
|  Error:
|  '.class' expected
|  int 1first
|      ^
|  Error:
|  not a statement
|  int 1first
|        ^--^
|  Error:
|  unexpected type
|    required: value
|    found:    class
|  int 1first
|  ^--^

jshell>
```

A.8 Show that braces do not have to occur in matching pairs inside a string literal.

Answer:

Click here to view code image

```
jshell> "Unmatched brace { in a string is OK"
$1 ==> "Unmatched brace { in a string is OK"

jshell>
```

A.9 Show what happens when you enter each of the following code snippets into JShell:

a. `System.out.println("seems OK")`

b. `System.out.println("missing something?)`

c. `System.out.println"missing something else?")`

Answer:

Click here to view code image

```
jshell> System.out.println("seems OK")
seems OK
```

```
jshell> System.out.println("missing something?)
|  Error:
|  unclosed string literal
|  System.out.println("missing something?)
|                     ^

jshell> System.out.println"missing something else?")
|  Error:
|  ';' expected
|  System.out.println"missing something else?")
|                    ^
|  Error:
|  cannot find symbol
|    symbol:   variable println
|  System.out.println"missing something else?")
|  ^-----------------^

jshell>
```

A.10 In a new session, demonstrate that after a `System.out.print` the next print results appear on the same line right after the previous one's. [*Hint:* To demonstrate this, reset the current session, enter two `System.out.print` statements, then use the following two commands to save the snippets to a file, then reload and re-execute them:

```
/save mysnippets
/open mysnippets
```

The `/open` command loads the `mysnippets` file's contents then executes them.]

Answer:

Click here to view code image

```
jshell> System.out.print("Happy ")
Happy
jshell> System.out.print("Birthday")
Birthday
jshell> /save mysnippets

jshell> /open mysnippets
Happy Birthday
jshell>
```

A.11 Demonstrate that after a `System.out.println`, the next text that prints displays its text at the left of the next line. [*Hint:* To demonstrate this, reset the current session, enter a

`System.out.println` statement followed by another print statement, then use the following two commands to save the snippets to a file, then reload and re-execute them:

```
/save mysnippets
/open mysnippets
```

The `/open` command loads the `mysnippets` file's contents then executes them.]

Answer:

Click here to view code image

```
jshell> System.out.println("Happy ")
Happy

jshell> System.out.println("Birthday")
Birthday

jshell> /save mysnippets

jshell> /open mysnippets
Happy
Birthday

jshell>
```

A.12 Demonstrate that you can reset a JShell session to remove all prior snippets and start from scratch without having to exit JShell and start a new session.

Answer:

```
jshell> int x = 10
x ==> 10

jshell> int y = 20
y ==> 20

jshell> x + y
$3 ==> 30

jshell> /reset
|  Resetting state.

jshell> /list

jshell>
```

A.13 Using `System.out.println`, demonstrate that the escape sequence `\n` causes a newline to be issued to the output. Use the string

```
"Welcome\nto\nJShell!"
```

Answer:

Click here to view code image

```
jshell> System.out.println("Welcome\nto\nJShell!")
Welcome
to
JShell!

jshell>
```

A.14 Demonstrate that the escape sequence \t causes a tab to be issued to the output. Note that your output will depend on how tabs are set on your system. Use the string

Click here to view code image

```
"before\tafter\nbefore\t\tafter"
```

Answer:

Click here to view code image

```
jshell> System.out.println("before\tafter\nbefore\t\tafter")
before  after
before          after

jshell>
```

A.15 Demonstrate what happens when you include a single backslash (\) in a string. Be sure that the character after the backslash does not create a valid escape sequence.

Answer:

Click here to view code image

```
jshell> System.out.println("Bad escap\e")
|  Error:
|  illegal escape character
|  System.out.println("Bad escap\e")
|                               ^
jshell>
```

A.16 Display a string containing `\\\\` (recall that `\\` is an escape sequence for a backslash). How many backslashes are displayed?

Answer: Two.

Click here to view code image

```
jshell> System.out.println("Displaying backslashes \\\\")
Displaying backslashes \\

jshell>
```

A.17 Use the escape sequence `\"` to display a quoted string.

Answer:

Click here to view code image

```
jshell> System.out.println("\"This is a string in quotes\"")
"This is a string in quotes"

jshell>
```

A.18 What happens when the following code executes in JShell:

Click here to view code image

```
System.out.println("Happy Birthday!\rSunny")
```

Answer: `\r` moves the output cursor back to the beginning of the current line, so `Sunny` overwrites `Happy`.

Click here to view code image

```
jshell> System.out.println("Happy Birthday!\rSunny")
Sunny Birthday!

jshell>
```

A.19 Consider the following statement:

Click here to view code image

```
System.out.printf("%s%n%s%n", "Welcome to ", "Java Programming!")
```

Make the following intentional errors (separately) to observe what happens.

a. Omit the parentheses around the argument list.

b. Omit the commas.

c. Omit one of the `%s%n` sequences.

d. Omit one of the strings (i.e., the second or the third argument).

e. Replace the first `%s` with `%d`.

f. Replace the string `"Welcome to "` with the integer `23`.

Answer:

a.

Click here to view code image

```
jshell> System.out.printf"%s%n%s%n", "Welcome to ", "Java Programm
|  Error:
|  ';' expected
|  System.out.printf"%s%n%s%n", "Welcome to ", "Java Programming!"
|                   ^
|  Error:
|  cannot find symbol
|    symbol:    variable printf
|  System.out.printf"%s%n%s%n", "Welcome to ", "Java Programming!"
|  ^---------------^

jshell>
```

b.

Click here to view code image

```
jshell> System.out.printf("%s%n%s%n" "Welcome to " "Java Programmi
|  Error:
|  ')' or ',' expected
|  System.out.printf("%s%n%s%n" "Welcome to " "Java Programming!")
|                                ^

jshell>
```

c.

Click here to view code image

```
jshell> System.out.printf("%s%n", "Welcome to ", "Java Programming
Welcome to
$6 ==> java.io.PrintStream@2038ae61

jshell>
```

d.

Click here to view code image

```
jshell> System.out.printf("%s%n%s%n", "Welcome to ")
Welcome to
|  Exception java.util.MissingFormatArgumentException: Format spec
'%s'
|        at Formatter.format (Formatter.java:2800)
|        at PrintStream.implFormat (PrintStream.java:1369)
|        at PrintStream.format (PrintStream.java:1348)
|        at PrintStream.printf (PrintStream.java:1247)
|        at (#7:1)

jshell>
```

e.

Click here to view code image

```
jshell> System.out.printf("%d%n%s%n", "Welcome to ", "Java Program
|  Exception java.util.IllegalFormatConversionException: d !=
java.lang.String
|        at Formatter$FormatSpecifier.failConversion
(Formatter.java:4710)
|        at Formatter$FormatSpecifier.printInteger (Formatter.java
|        at Formatter$FormatSpecifier.print (Formatter.java:3203)
|        at Formatter.format (Formatter.java:2801)
|        at PrintStream.implFormat (PrintStream.java:1369)
|        at PrintStream.format (PrintStream.java:1348)
|        at PrintStream.printf (PrintStream.java:1247)
|        at (#8:1)

jshell>
```

f.

Click here to view code image

```
jshell> System.out.printf("%s%n%s%n", 23, "Java Programming!")
23
Java Programming!
$9 ==> java.io.PrintStream@2038ae61

jshell>
```

A.20 What happens when you enter the `/imports` command in a new JShell session?

Answer:

Click here to view code image

```
jshell> /imports
|    import java.io.*
|    import java.math.*
|    import java.net.*
|    import java.nio.file.*
|    import java.util.*
|    import java.util.concurrent.*
|    import java.util.function.*
|    import java.util.prefs.*
|    import java.util.regex.*
|    import java.util.stream.*

jshell>
```

A.21 Import class `Scanner`, then create a `Scanner` object `input` for reading from `System.in`. What happens when you execute the statement:

```
int number = input.nextInt()
```

and the user enters the string `"hello"`?
Answer:

Click here to view code image

```
jshell> import java.util.Scanner

jshell> var input = new Scanner(System.in)
input ==> java.util.Scanner[delimiters=\p{javaWhitespace}+] ...
\E][infinity string=\Q∞\E]
jshell> int number = input.nextInt()
hello
|  Exception java.util.InputMismatchException
|        at Scanner.throwFor (Scanner.java:964)
|        at Scanner.next (Scanner.java:1619)
|        at Scanner.nextInt (Scanner.java:2284)
|        at Scanner.nextInt (Scanner.java:2238)
|        at (#12:1)
jshell>
```

A.22 In a new or `/reset` JShell session, repeat Exercise A.21 without importing class `Scanner` to demonstrate that the package `java.util` is already imported in JShell.

Answer:

Click here to view code image

```
jshell> var input = new Scanner(System.in)
input ==> java.util.Scanner[delimiters=\p{javaWhitespace}+] ...
\E][infinity string=\Q∞\E]

jshell> int number = input.nextInt()
hello
|  Exception java.util.InputMismatchException
|        at Scanner.throwFor (Scanner.java:964)
|        at Scanner.next (Scanner.java:1619)
|        at Scanner.nextInt (Scanner.java:2284)
|        at Scanner.nextInt (Scanner.java:2238)
|        at (#2:1)

jshell>
```

A.23 Demonstrate what happens when you don't precede a `Scanner` input operation with a meaningful prompting message telling the user what to input. Enter the following statements:

Click here to view code image

```
var input = new Scanner(System.in)
int value = input.nextInt()
```

Answer: JShell appears to hang while it waits for the user to enter a value and press (or select) *Enter*.

Click here to view code image

```
jshell> var input = new Scanner(System.in)
input ==> java.util.Scanner[delimiters=\p{javaWhitespace}+] ...
\E][infinity string=\Q∞\E]

jshell> int value = input.nextInt()
```

A.24 Demonstrate that you can't place an `import` statement in a class.

Answer:

Click here to view code image

```
jshell> class Demonstration {
   ...>    import java.util.Scanner;
   ...> }
|  Error:
|  illegal start of type
|     import java.util.Scanner;
|     ^
|  Error:
```

```
|  <identifier> expected
|      import java.util.Scanner;
|                              ^

jshell> import java.util.Scanner

jshell> class Demonstration {
   ...> }
|  created class Demonstration

jshell>
```

A.25 In a new session, demonstrate that identifiers are case sensitive by declaring variables `id` and `ID` of types `String` and `int`, respectively. Also use the `/list` command to output the two snippets representing the separate variables.

Answer:

```
jshell> String id = "Natasha"
id ==> "Natasha"

jshell> int ID = 413
ID ==> 413

jshell> /list

   1 : String id = "Natasha";
   2 : int ID = 413;

jshell>
```

A.26 Demonstrate that initialization statements like

```
String month = "April"
int age = 65
```

indeed initialize their variables with the indicated values.

Answer:

Click here to view code image

```
jshell> String month = "April"
month ==> "April"

jshell> month
month ==> "April"

jshell> int age = 65
age ==> 65
```

```
jshell> age
age ==> 65

jshell> /vars
|    String month = "April"
|    int age = 65

jshell>
```

A.27 Demonstrate what happens when you:

a. Add 1 to the largest possible `int` value 2,147,483,647.

b. Subtract 1 from the smallest possible integer – 2,147,483,648.

Answer:

```
jshell> 2147483647 + 1
$1 ==> -2147483648

jshell> -2147483648 - 1
$2 ==> 2147483647

jshell>
```

A.28 Demonstrate that large integers like `1234567890` are equivalent to their counterparts with the underscore separators, namely `1_234_567_890`:

a. `1234567890 == 1_234_567_890`

b. Print each of these values to confirm that you get the same result.

c. Divide each of these values by 2 to confirm that you get the same result.

Answer:

Click here to view code image

```
jshell> 1234567890 == 1_234_567_890
$1 ==> true

jshell> System.out.println(1234567890)
1234567890

jshell> System.out.println(1_234_567_890)
1234567890
```

```
jshell> 1234567890 / 2
$4 ==> 617283945

jshell> 1_234_567_890 / 2
$5 ==> 617283945

jshell>
```

A.29 Placing spaces around operators in an arithmetic expression does not affect the value of that expression. In particular, the following expressions are equivalent:

```
17+23
17 + 23
```

Demonstrate this with the condition

```
(17+23) == (17 + 23)
```

Answer:

```
jshell> (17+23) == (17 + 23)
$1 ==> true

jshell>
```

A.30 Demonstrate that the parentheses around the argument `number1 + number2` in the following statement are unnecessary:

Click here to view code image

```
System.out.printf("Sum is %d%n", (number1 + number2))
```

Answer:

Click here to view code image

```
jshell> int number1 = 10
number1 ==> 10

jshell> int number2 = 20
number2 ==> 20

jshell> System.out.printf("Sum is %d%n", (number1 + number2))
Sum is 30
$3 ==> java.io.PrintStream@23e028a9

jshell> System.out.printf("Sum is %d%n", number1 + number2)
```

```
Sum is 30
$4 ==> java.io.PrintStream@23e028a9

jshell>
```

A.31 Declare the `int` variable `x` and initialize it to `14`, then demonstrate that the subsequent assignment `x = 27` overwrites `x`'s previous value.

Answer:

```
jshell> int x = 14
x ==> 14

jshell> x = 27
x ==> 27

jshell>
```

A.32 Demonstrate that printing the value of the following variable does not modify its value:

```
int y = 29
```

Answer:

```
jshell> int y = 29
y ==> 29

jshell> System.out.println(y)
29

jshell> y
y ==> 29
```

A.33 Using the declarations:

```
int b = 7
int m = 9
```

a. Demonstrate that attempting to do algebraic multiplication by placing the variable names next to one another as in `bm` doesn't work in Java.

b. Demonstrate that the Java expression `b * m` indeed multiplies the two operands.

Answer:

```
jshell> int b = 7
b ==> 7

jshell> int m = 9
m ==> 9

jshell> bm
|  Error:
|  cannot find symbol
|    symbol:   variable bm
|  bm
|  ^^

jshell> b * m
$3 ==> 63

jshell>
```

A.34 Use the following expressions to demonstrate that integer division yields an integer result:

a. `8 / 4`

b. `7 / 5`

Answer: a) 2. b) 1.

```
jshell> 8 / 4
$1 ==> 2

jshell> 7 / 5
$2 ==> 1

jshell>
```

A.35 Demonstrate what happens when you attempt each of the following integer divisions:

a. `0 / 1`

b. `1 / 0`

c. `0 / 0`

Answer:

Click here to view code image

```
jshell> 0 / 1
$1 ==> 0

jshell> 1 / 0
```

```
|  java.lang.ArithmeticException thrown: / by zero
|        at (#2:1)

jshell> 0 / 0
|  java.lang.ArithmeticException thrown: / by zero
|        at (#3:1)

jshell>
```

A.36 Demonstrate that the values of the following expressions:

a. `(3 + 4 + 5) / 5`

b. `3 + 4 + 5 / 5`

are different and thus the parentheses in the first expression are required if you want to divide the entire quantity 3 + 4 + 5 by 5.
Answer:

```
jshell> (3 + 4 + 5) / 5
$1 ==> 2

jshell> 3 + 4 + 5 / 5
$2 ==> 8

jshell>
```

A.37 Calculate the value of the following expression:

```
5 / 2 * 2 + 4 % 3 + 9 - 3
```

manually being careful to observe the rules of operator precedence. Confirm the result in JShell.
Answer:

Click here to view code image

```
jshell> 5 / 2 * 2 + 4 % 3 + 9 - 3
$11 ==> 11

jshell>
```

A.38 Test each of the two equality and four relational operators on the two values 7 and 7. For example, `7 == 7`, `7 < 7`, etc.

Answer:

```
jshell> 7 == 7
$12 ==> true
```

```
jshell> 7 != 7
$13 ==> false

jshell> 7 < 7
$14 ==> false

jshell> 7 <= 7
$15 ==> true

jshell> 7 > 7
$16 ==> false

jshell> 7 >= 7
$17 ==> true

jshell>
```

A.39 Repeat Exercise A.38 using the values `7` and `9`.

Answer:

```
jshell> 7 == 9
$18 ==> false

jshell> 7 != 9
$19 ==> true

jshell> 7 < 9
$20 ==> true

jshell> 7 <= 9
$21 ==> true

jshell> 7 > 9
$22 ==> false

jshell> 7 >= 9
$23 ==> false

jshell>
```

A.40 Repeat Exercise A.38 using the values `11` and `9`.

Answer:

```
jshell> 11 == 9
$24 ==> false

jshell> 11 != 9
$25 ==> true
```

```
jshell> 11 < 9
$26 ==> false

jshell> 11 <= 9
$27 ==> false

jshell> 11 > 9
$28 ==> true

jshell> 11 >= 9
$29 ==> true

jshell>
```

A.41 Demonstrate that accidentally placing a semicolon after the right parenthesis of the condition in an if statement can be a logic error.

Click here to view code image

```
if (3 == 5); {
   System.out.println("3 is equal to 5");
}
```

Answer:

Click here to view code image

```
jshell> if (3 == 5); {
   ...>    System.out.println("3 is equal to 5");
   ...> }
3 is equal to 5

jshell>
```

A.42 Given the following declarations:

```
int x = 1
int y = 2
int z = 3
int a
```

what are the values of `a`, `x`, `y` and `z` after the following statement executes?

```
a = x = y = z = 10
```

Answer:

```
jshell> int x = 1
x ==> 1

jshell> int y = 2
y ==> 2

jshell> int z = 3
z ==> 3

jshell> int a
a ==> 0

jshell> a = x = y = z = 10
a ==> 10

jshell> x
x ==> 10

jshell> y
y ==> 10

jshell> z
z ==> 10

jshell>
```

A.43 Manually determine the value of the following expression then use JShell to check your work:

```
(3 * 9 * (3 + (9 * 3 / (3))))
```

Answer:

Click here to view code image

```
jshell> (3 * 9 * (3 + (9 * 3 / (3))))
$42 ==> 324

jshell>
```

Formatted Output

Objectives

In this appendix, you'll:

- Use `printf` to format outputs.
- Format decimal, octal and hexadecimal integers.
- Format floating-point numbers with fixed-point and scientific notations.
- Format `String`s and characters.
- Format with field widths and precisions.
- Use various `printf` formatting flags to align data, display integer bases, pad fields with 0s, display numeric values with their locale-specific thousands separator and display negative values in parentheses.
- Use argument indexes to specify where to place each argument in a format string.
- Format `String`s in memory rather than displaying formatted output.

Outline

B.1 Introduction

B.2 Output with `printf`

B.3 Integer Formatting

B.4 Floating-Point Number Formatting

B.5 String and Character Formatting

B.6 Other Conversion Characters

B.7 Field Widths and Precisions

B.8 Formatting Flags

B.9 Argument Indices for Explicit Positioning in Format Strings

B.10 Escape Sequences

B.11 Formatting `String`s in Memory

B.12 Wrap-Up

Special Section: Generative AI Exercises

B.1 Introduction

This appendix presents formatting features of method `printf` and class `Formatter` (package `java.util`). Class `Formatter` formats data as a `String` in memory by default but also can send formatted output to a specified destination, such as a file. Method `printf` uses a `Formatter` object to send formatted output to the standard output. We've discussed many `printf` features throughout the book. Here, we summarize those features and introduce others, such as using argument indexes to position arguments in a format string, and displaying numbers and `String`s with various flags. We briefly introduced date/time formatting in Section 5.12, Objects-Natural Case Study: Java Date/Time API. You'll explore date/time formatting further in Generative AI Exercise B.2.

B.2 Output with `printf`

Precise output formatting is accomplished with `printf`. Java borrowed (and enhanced) this feature from the C programming language. Method `printf` can perform the following formatting capabilities, each of which is discussed in this appendix:

- Rounding floating-point values to an indicated number of decimal places.
- Aligning a column of numbers with decimal points appearing one above the other.
- Right alignment and left alignment of outputs.
- Inserting literal characters at precise locations in a line of output.
- Representing floating-point numbers in exponential format.
- Representing integers in octal and hexadecimal format.
- Displaying all types of data with fixed-size field widths and precisions.

Method `printf` also can display dates and times, but the class `DateTimeFormatter` (introduced in Section 5.12) is preferred.

Every call to `printf` supplies as the first argument a format string that describes the output format. The format string may consist of fixed text and format specifiers. Fixed text is output by `printf` just as it would be output by `System.out` methods `print` or `println`. Each format specifier is a placeholder for a value and specifies the data type to output. Format specifiers may also include

optional formatting information.

In the simplest form, each format specifier begins with a percent sign (`%`), followed by a **conversion character** representing the data type of the value to output. For example, the format specifier `%s` is a placeholder for a `String`, and the format specifier `%d` is a placeholder for an `int` value. The optional formatting information, such as an argument index, flags, field width and precision, is specified between the percent sign and the conversion character. We demonstrate each of these capabilities.

B.3 Integer Formatting

The following table describes the **integer conversion characters**—Appendix C overviews the binary, octal, decimal and hexadecimal number systems.

Conversion character	Description
d	Display a decimal (base 10) integer.
o	Display an octal (base 8) integer.
x or **X**	Display a hexadecimal (base 16) integer. `X` uses uppercase letters.

Figure B.1 uses each to print an integer. The plus sign is not displayed by default, but the minus sign is. Later in this appendix (Fig. B.8), we'll see how to force plus signs to print.

Click here to view code image

```
1  // Fig. B.1: IntegerConversionTest.java
2  // Using the integer conversion characters.
3
4  public class IntegerConversionTest {
5     public static void main(String[] args) {
6        System.out.printf("%d%n", 26);
7        System.out.printf("%d%n", +26);
8        System.out.printf("%d%n", -26);
9        System.out.printf("%o%n", 26);
10       System.out.printf("%x%n", 26);
11       System.out.printf("%X%n", 26);
12    }
13 }
```

```
26
```

```
26
-26
32
1a
1A
```

Fig. B.1 | Using the integer conversion characters.

The `printf` method has the form

Click here to view code image

```
printf(format-string, argument-list);
```

where *format-string* describes the output format and the optional *argument-list* contains the values corresponding to each format specifier in *format-string*. There can be many format specifiers in one format string.

Each format string in lines 6–8 specifies that `printf` should output a decimal integer (`%d`) followed by a newline character. At the format specifier's position, `printf` substitutes the first argument's value after the format string. If the format string contains multiple format specifiers, at each subsequent format specifier's position `printf` substitutes the value of the next argument in the argument list. The `%o` format specifier in line 9 outputs the integer in octal format. The `%x` format specifier in line 10 outputs the integer in hexadecimal format with lowercase letters. The `%X` format specifier in line 11 outputs the integer in hexadecimal format with uppercase letters.

B.4 Floating-Point Number Formatting

The following table describes the floating-point conversions:

Conversion character	Description
`e` or `E`	Display a floating-point value in exponential notation. Conversion character `E` displays the output in uppercase letters.
`f` or `F`	Display a floating-point value in fixed-point format.
`g` or `G`	Display a floating-point value in either the fixed-point format `f` or the exponential format `e` based on the magnitude of the value. If the magnitude is less than 10^{-3} or greater than or equal to 10^{7}, the floating-point value is printed with `e` (or `E`). Otherwise, the value is printed in format `f`. The output is displayed in uppercase letters when conversion character `G` is used.

Display a floating-point number in hexadecimal format. Conversion character `A` displays the output in uppercase letters.

The **conversion characters** `e` and `E` display floating-point values in **computerized scientific notation**, also called **exponential notation**. Exponential notation is the computer equivalent of the scientific notation used in mathematics. For example, the value 150.4582 is represented in scientific notation in mathematics as

$$1.504582 \times 10^2$$

and is represented in exponential notation as

```
1.504582e+02
```

in Java. This notation indicates that 1.504582 is multiplied by 10, raised to the second power `(e+02)`. The e stands for "exponent."

Values printed with the conversion characters e, `E` and `f` are output with six digits of precision to the right of the decimal point by default (e.g., `1.045921`)—other precisions must be specified explicitly. For values printed with the conversion character `g`, the precision represents the total number of digits displayed, excluding the exponent. The default is six digits (e.g., `12345678.9` is displayed as `1.23457e+07`). **Conversion character** `f` always prints at least one digit to the left of the decimal point. Conversion characters `e` and `E` print lowercase `e` and uppercase `E` preceding the exponent and always print exactly one digit to the left of the decimal point. Rounding occurs if the value being formatted has more significant digits than the precision.

Conversion character `g` (or `G`) prints in either `e` (`E`) or `f` format, depending on the floating-point value. For example, the values `0.0000875`, `87500000.0`, `8.75`, `87.50` and `875.0` are printed as `8.750000e-05`, `8.750000e+07`, `8.750000`, `87.500000` and `875.000000` with the conversion character `g`. The value `0.0000875` uses `e` notation because the magnitude is less than 10^{-3}. The value `87500000.0` uses `e` notation because the magnitude is greater than 10^7. Figure B.2 demonstrates the floating-point conversion characters.

Click here to view code image

```
// Fig. B.2: FloatingNumberTest.java
// Using floating-point conversion characters.

```

```
4  public class FloatingNumberTest {
5     public static void main(String[] args) {
6        System.out.printf("%e%n", 12345678.9);
7        System.out.printf("%e%n", +12345678.9 );
8        System.out.printf("%e%n", -12345678.9);
9        System.out.printf("%E%n", 12345678.9);
10       System.out.printf("%f%n", 12345678.9);
11       System.out.printf("%g%n", 12345678.9);
12       System.out.printf("%G%n", 12345678.9);
13    }
14 }
```

```
1.234568e+07
1.234568e+07
-1.234568e+07
1.234568E+07
12345678.900000
1.23457e+07
1.23457E+07
```

Fig. B.2 | Using floating-point conversion characters.

B.5 String and Character Formatting

Conversion characters `c` and `s` print individual characters and `String`s, respectively. **Conversion characters `c` and `C`** require a `char` argument. **Conversion characters `s` and `S`** can take a `String` or any `Object` as an argument—any `Object` can be converted to a `String` because they all have `toString` methods. When conversion characters `C` and `S` are used, the output is displayed in uppercase letters. Figure B.3 displays characters, `String`s and objects with conversion characters `c`, `s` and `S`. Autoboxing occurs at line 7 when an `int` constant is assigned to an `Integer` object. Line 13 outputs an `Integer` argument with the conversion character `s`, which implicitly invokes the `toString` method to get the integer value. You can also output an `Integer` object using the `%d` format specifier. In this case, the `int` value in the `Integer` object will be unboxed and output. Using `%c` to print a `String` causes an `IllegalFormatConversionException`—a `String` cannot be converted to a character.

Click here to view code image

```
1  // Fig. B.3: CharStringConversion.java
```

```
2  // Using character and string conversion characters.
3  public class CharStringConversion {
4     public static void main(String[] args) {
5        char character = 'A'; // initialize char
6        String string = "This is also a string"; // String object
7        Integer integer = 1234; // initialize integer (autoboxing)
8
9        System.out.printf("%c%n", character);
10       System.out.printf("%s%n", "This is a string");
11       System.out.printf("%s%n", string);
12       System.out.printf("%S%n", string);
13       System.out.printf("%s%n", integer); // implicit call to toString
14    }
15 }
```

```
A
This is a string
This is also a string
THIS IS ALSO A STRING
1234
```

Fig. B.3 | Using character and `String` conversion characters.

B.6 Other Conversion Characters

The following table describes the remaining conversion characters—`b`, `B`, `h`, `H`, `%` and `n`:

Conversion character	Description
`b` or `B`	Print `"true"` or `"false"` for the value of a `boolean` or `Boolean`. These conversion characters can also format the value of any reference. If the reference is non-`null`, `"true"` is output; otherwise, `"false"`. When conversion character `B` is used, the output is displayed in uppercase letters.
`h` or `H`	Print the `String` representation of an object's hash-code value in hexadecimal format. If the corresponding argument is `null`, `"null"` is printed. When conversion character `H` is used, the output is displayed in uppercase letters.
`%`	Print the percent character.
`n`	Print the platform-specific line separator (e.g., `\r\n` on Windows or `\n` on macOS/Linux).

Lines 7–8 of Fig. B.4 use `%b` to print the value of `boolean` (or

`Boolean`) values `false` and `true`. Line 9 associates a `String` to `%b`, which returns `true` because it's not `null`. Line 10 associates a `null` object to `%B`, which displays `FALSE` because `test` is `null`. Lines 11–12 use `%h` to print the `String` representations of the hash-code values for `String`s `"hello"` and `"Hello"`. These values could be used to store or locate the `String`s in a `HashMap` (Chapter 12, Generic Collections). The hash-code values for these two `String`s differ because one `String` starts with a lowercase letter and the other with an uppercase letter. Line 13 uses `%H` to print `null` in uppercase letters. The last two `printf` statements (lines 14–15) use `%%` to print the `%` character in a `String` and `%n` to print a platform-specific line separator.

Click here to view code image

```
1  // Fig. B.4: OtherConversion.java
2  // Using the b, B, h, H, % and n conversion characters.
3
4  public class OtherConversion {
5     public static void main(String[] args) {
6        Object test = null;
7        System.out.printf("%b%n", false);
8        System.out.printf("%b%n", true);
9        System.out.printf("%b%n", "Test");
10       System.out.printf("%B%n", test);
11       System.out.printf("Hashcode of \"hello\" is %h%n", "hello");
12       System.out.printf("Hashcode of \"Hello\" is %h%n", "Hello");
13       System.out.printf("Hashcode of null is %H%n", test);
14       System.out.printf("Printing a %% in a format string%n");
15       System.out.printf("Printing a new line %nnext line starts here");
16    }
17 }
```

```
false
true
true
FALSE
Hashcode of "hello" is 5e918d2
Hashcode of "Hello" is 42628b2
Hashcode of null is NULL
Printing a % in a format string
Printing a new line
next line starts here
```

Fig. B.4 | Using the `b`, `B`, `h`, `H`, `%` and `n` conversion characters.

Trying to print a literal percent character using `%` rather than `%%`

in the format string might cause a difficult-to-detect logic error. When `%` appears in a format string, it must be followed by a conversion character in the `String`. The single percent could accidentally be followed by a legitimate conversion character, thus causing a logic error.

B.7 Field Widths and Precisions

The size of a field in which data is printed is specified by a **field width**. If the field width is larger than the data being printed, the data is right aligned in that field by default. We discuss left alignment in Section B.8. You insert an integer representing the field width between the `%` and the conversion character (e.g., `%4d`) in the format specifier. Figure B.5 prints two groups of five numbers each, right aligning those numbers that contain fewer digits than the field width. The field width is increased to print values wider than the field. The minus sign for a negative value uses one character position in the field. Also, if no field width is specified, the data prints in exactly as many positions as it needs. Field widths can be used with all format specifiers except the line separator (`%n`). Not providing a sufficiently large field width to handle a value to be printed can offset other data being printed and produce confusing outputs. Know your data!

Click here to view code image

```
1  // Fig. B.5: FieldWidthTest.java
2  // Right aligning integers in fields.
3
4  public class FieldWidthTest {
5     public static void main(String[] args) {
6        System.out.printf("%4d%n", 1);
7        System.out.printf("%4d%n", 12);
8        System.out.printf("%4d%n", 123);
9        System.out.printf("%4d%n", 1234);
10       System.out.printf("%4d%n%n", 12345); // data too large
11
12       System.out.printf("%4d%n", -1);
13       System.out.printf("%4d%n", -12);
14       System.out.printf("%4d%n", -123);
15       System.out.printf("%4d%n", -1234); // data too large
16       System.out.printf("%4d%n", -12345); // data too large
17    }
18 }
```

```
    1
   12
  123
 1234
12345

   -1
  -12
 -123
-1234
-12345
```

Fig. B.5 | Right aligning integers in fields.

Method `printf` also provides the ability to specify the precision with which data is printed. Precision has different meanings for different types. When used with floating-point conversion characters `e` and `f`, the precision is the number of digits that appear after the decimal point. When used with conversion characters `g`, `a` or `A`, the precision is the maximum number of significant digits to be printed. When used with conversion character `s`, the precision is the maximum number of characters to be written from the `String`. To use precision, place between the percent sign and the conversion specifier a decimal point (`.`) followed by an integer representing the precision. Figure B.6 demonstrates precision in format strings. When a floating-point value is printed with a precision smaller than the original value's number of decimal places, the value is rounded. Also, the format specifier `%.3g` indicates that the total number of digits used to display the floating-point value is 3. Because the value has three digits to the left of the decimal point, it's rounded to the ones position.

Click here to view code image

```
1  // Fig. B.6: PrecisionTest.java
2  // Using precision for floating-point numbers and strings.
3  public class PrecisionTest {
4     public static void main(String[] args) {
5        double f = 123.94536;
6        String s = "Happy Birthday";
7
8        System.out.printf("Using precision for floating-point numbers%n");
9        System.out.printf("\t%.3f%n\t%.3e%n\t%.3g%n%n", f, f, f);
10
11       System.out.printf("Using precision for strings%n");
12       System.out.printf("\t%.11s%n", s);
```

```
13        }
14     }
```

```
Using precision for floating-point numbers
        123.945
        1.239e+02
        124

Using precision for strings
        Happy Birth
```

Fig. B.6 | Using precision for floating-point numbers and `String`s.

The field width and the precision can be combined by placing the field width, followed by a decimal point, followed by a precision between the percent sign and the conversion character, as in the statement

```
printf("%9.3f", 123.456789);
```

which displays `123.457` with three digits to the right of the decimal point right aligned in a nine-digit field—this number will be preceded in its field by two blanks.

B.8 Formatting Flags

Various flags may be used with method `printf` to supplement its output formatting capabilities. The following flags are available for use in format strings:

Flag	Description
`-`	Left align the output within the specified field.
`+`	Display a plus sign preceding positive values and a minus sign preceding negative values.
space	Print a space before a positive value not printed with the `+` flag.
`#`	When used with the octal conversion character `o`, prefix `0` to the output value. When used with the hexadecimal conversion character `x`, prefix `0x` to the output value.
`0` (zero)	Pad a field with leading zeros.
`,` (comma)	Use the locale-specific thousands separator (i.e., `','` for U.S. locale) to display decimal and floating-point numbers.
`(`	Enclose negative numbers in parentheses.

To use a flag in a format string, place it immediately to the right of the percent sign. Several flags may be used in the same format specifier. Figure B.7 demonstrates right alignment and left alignment of a `String`, an integer, a character and a floating-point number. Line 7 serves as a counting mechanism for the screen output.

Click here to view code image

```
1  // Fig. B.7: MinusFlagTest.java
2  // Right aligning and left aligning values.
3
4  public class MinusFlagTest {
5     public static void main(String[] args) {
6        System.out.println("Columns:");
7        System.out.println("01234567890123456789012345678901234567890123456789\n");
8        System.out.printf("%10s%10d%10c%10f%n%n", "hello", 7, 'a', 1.23);
9        System.out.printf(
10          "%-10s%-10d%-10c%-10f%n", "hello", 7, 'a', 1.23);
11    }
12 }
```

```
Columns:
01234567890123456789012345678901234567890123456789

     hello         7         a  1.230000

hello     7         a         1.230000
```

Fig. B.7 | Right aligning and left aligning values.

Figure B.8 prints a positive and a negative number, each with and without the **+ flag**. The minus sign is displayed in both cases, and the plus sign only when the `+` flag is used.

Click here to view code image

```
1  // Fig. B.8: PlusFlagTest.java
2  // Printing numbers with and without the + flag.
3
4  public class PlusFlagTest {
5     public static void main(String[] args) {
6        System.out.printf("%d\t%d%n", 786, -786);
7        System.out.printf("%+d\t%+d%n", 786, -786);
8  }
9  }
```

```
786     -786
+786    -786
```

Fig. B.8 | Printing numbers with and without the `+` flag.

Figure B.9 prefixes a space to the positive number with the **space flag**—useful for aligning positive and negative numbers with the same number of digits. The value `-547` is not preceded by a space in the output because of its minus sign.

Click here to view code image

```
1  // Fig. B.9: SpaceFlagTest.java
2  // Printing a space before non-negative values.
3
4  public class SpaceFlagTest {
5     public static void main(String[] args) {
6        System.out.printf("% d%n% d%n", 547, -547);
7     }
8  }
```

```
 547
-547
```

Fig. B.9 | Printing a space before nonnegative values.

Figure B.10 uses the **`#` flag** to prefix `0` to the octal value and 0x to the hexadecimal value.

Click here to view code image

```
1  // Fig. B.10: PoundFlagTest.java
2  // Using the # flag with conversion characters o and x.
3
4  public class PoundFlagTest {
5     public static void main(String[] args) {
6        int c = 31; // initialize c
7
8        System.out.printf("%#o%n", c);
9        System.out.printf("%#x%n", c);
10    }
11 }
```

```
037
0x1f
```

Fig. B.10 | Using the `#` flag with conversion characters `o` and `x`.

Figure B.11 combines the `+` flag, the `0` **flag** and the space flag to print `452` in a field of width 9 with a `+` sign and leading zeros, next prints `452` in a field of width 9 using only the `0` flag, then prints `452` in a field of width 9 using only the space flag.

Click here to view code image

```
1  // Fig. B.11: ZeroFlagTest.java
2  // Printing with the 0 (zero) flag fills in leading zeros.
3
4  public class ZeroFlagTest {
5     public static void main(String[] args) {
6        System.out.printf("%+09d%n", 452);
7        System.out.printf("%09d%n", 452);
8        System.out.printf("% 9d%n", 452);
9     }
10 }
```

```
+00000452
000000452
      452
```

Fig. B.11 | Printing with the `0` (zero) flag fills in leading zeros.

Figure B.12 uses the comma (`,`) flag to display a decimal and a floating-point number with the thousands separator.

Click here to view code image

```
1  // Fig. B.12: CommaFlagTest.java
2  // Using the comma (,) flag to display numbers with thousands separator.
3
4  public class CommaFlagTest {
5     public static void main(String[] args) {
6        System.out.printf("%,d%n", 58625);
7        System.out.printf("%,.2f%n", 58625.21);
8        System.out.printf("%,.2f%n", 12345678.9);
9     }
10 }
```

```
58,625
58,625.21
12,345,678.90
```

Fig. B.12 | Using the comma (`,`) flag to display numbers with the thousands separator.

Figure B.13 encloses negative numbers in parentheses using the `(` flag. The value `50` is not enclosed in parentheses in the output because it's a positive number.

Click here to view code image

```
1  // Fig. B.13: ParenthesesFlagTest.java
2  // Using the ( flag to place parentheses around negative numbers.
3
4  public class ParenthesesFlagTest {
5     public static void main(String[] args) {
6        System.out.printf("%(d%n", 50);
7        System.out.printf("%(d%n", -50);
8        System.out.printf("%(.1e%n", -50.0);
9  }
10 }
```

```
50
(50)
(5.0e+01)
```

Fig. B.13 | Using the `(` flag to place parentheses around negative numbers.

B.9 Argument Indices for Explicit Positioning in Format Strings

An **argument index** is an optional integer followed by a dollar sign (`$`) indicating the argument's position in the argument list. Argument indices enable programmers to specify the exact position of each argument. For software that supports multiple languages, reordering arguments is useful for translated format strings, because spoken languages often order words differently. Argument indices enable you to use different format strings without changing the order of `printf`'s other arguments. Argument indices also help avoid duplicating arguments. Figure B.14 prints arguments in the argument list in reverse order using the argument index.

Click here to view code image

```
1  // Fig. B.14: ArgumentIndexTest.java
```

```
// Reordering output with argument indices.

public class ArgumentIndexTest {
   public static void main(String[] args) {
      System.out.printf(
         "Parameter list without reordering: %s %s %s %s%n",
         "first", "second", "third", "fourth");
      System.out.printf(
         "Parameter list after reordering: %4$s %3$s %2$s %1$s%n",
         "first", "second", "third", "fourth");
   }
}
```

```
Parameter list without reordering: first second third fourth
Parameter list after reordering: fourth third second first
```

Fig. B.14 | Reordering output with argument indices.

B.10 Escape Sequences

Most literal characters you output with `printf` can simply be included in the format string. However, there are several "problem" characters, such as the quotation marks (`"`) that delimit the format string itself. Various control characters, such as newline and tab, must be represented by escape sequences. The following table lists the escape sequences and the actions they cause:

Escape sequence	Description
`\'`	Single quote (') character.
`\"`	Double quote (") character.
`\\`	Backslash (\) character.
`\b`	Move the cursor back one position on the current line.
`\f`	Move the cursor to the start of the next page.
`\n`	Move the cursor to the beginning of the next line.
`\r`	Move the cursor to the beginning of the current line. Subsequent outputs overwrite the current line.
`\t`	Move the cursor to the next horizontal tab position.

B.11 Formatting `String`s in Memory

So far, we've displayed formatted output. What if we want to create formatted `String`s rather than display formatted text? **Class `Formatter`** (package `java.util`) enables you to format `String`s in memory or send formatted `String`s to specified destinations, such as a file on disk. The `printf` method uses a `Formatter` object to provide its formatted output capabilities. By default, a `Formatter` creates a `String` in memory. Figure B.15 uses a `Formatter` to build a formatted `String` in memory, which we then output.

Click here to view code image

```
1  // Fig. B.15: FormatterTest.java
2  // Formatting output with class Formatter.
3  import java.util.Formatter;
4
5  public class FormatterTest {
6     public static void main(String[] args) {
7        // create Formatter and format output
8        Formatter formatter = new Formatter();
9        formatter.format("%d = %#o = %#X", 10, 10, 10);
10
11       System.out.println(formatter.toString());
12    }
13 }
```

```
10 = 012 = 0XA
```

Fig. B.15 | Formatting output with class `Formatter`.

Line 8 creates a `Formatter` object using the default constructor, so this object will build a `String` in memory. Other constructors are provided to allow you to specify the destination to which the formatted data should be output. For details, visit

Click here to view code image

```
https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/
   util/Formatter.html
```

Line 9 invokes **method `format`** to format the output. Like `printf`, `format` receives a format string and an argument list but `format` sends the formatted output to the destination specified by its constructor—a `String` in memory by default. Line 11 invokes the `Formatter`'s `toString` method to get the formatted data as a `String`, which we then display.

String `static` Method `format` and Instance Method `formatted`

Class `String` also provides two methods that simplify creating `String`s in memory:

- **`String static` method `format`** receives a format string and the values to format, and returns a formatted `String`. For example, the following statement creates the same formatted `String` as line 9:

  Click here to view code image

  ```
  String s = String.format("%d = %#o = %#x", 10, 10, 10);
  ```

- A string literal is a `String`, so you can use it to call `String` methods. **`String` method `formatted`** is similar to `format` but is called directly on a format string. You pass it only the values to format, as in:

  Click here to view code image

  ```
  String s = "%d = %#o = %#x".formatted(10, 10, 10);
  ```

`String` method `formatted` can also be called on text blocks. For example, the following statement uses a text block as a format string:

Click here to view code image

```
String book = """
Title: "%s"
Edition: %d
Copyright: %d""".formatted("Java How to Program", 12, 2025);
```

Printing the `book` formatted text block displays the following text:

```
    Title: "Java How to Program"
  Edition: 12
Copyright: 2025
```

B.12 Wrap-Up

This appendix summarized how to display formatted output and create formatted `String`s in memory using various format characters and flags. We output integers using format characters `d`,

`o`, `x` and `X`; and floating-point numbers using format characters `e`, `E`, `f`, `g` and `G`. We formatted text with field widths and precisions. We formatted numeric values by combining the numeric format characters with the flags `+`, `-`, space, `#`, `0`, comma and `(`. We used argument indices to specify where each argument should be placed in a format string. Finally, we formatted `String`s and text blocks in memory, first using class `Formatter` and its `format` method, then more conveniently using class `String` and its `format` and `formatted` methods.

Special Section: Generative AI Exercises

For each of the following questions, try multiple genAIs to determine which performs best. Always verify the results. Keep in mind that they sometimes make mistakes and even hallucinate, creating unreliable information.

B.1 ***(Generative AI: Format String Safety)*** Prompt genAIs asking them to explain why a Java program should never supply a user-input `String` as a format `String` to `printf`, a `Formatter`'s `format` method, or the `String` methods `format` and `formatted`. Ask for simple code examples showing how to prevent the problems they describe.

B.2 ***(Generative AI: Formatting Dates and Times)*** Prompt genAIs to create a code-based tutorial showing Java's most commonly used `DateTimeFormatter` capabilities.

B.3 ***(Generative AI: Locale-Specific Formatting)*** Prompt genAIs to create a code-based tutorial on locale-specific formatting in Java.

B.4 ***(Generative AI: Displaying Negative Values in Parentheses)*** Prompt genAIs asking why you'd want to display a negative value in parentheses rather than preceding it with a minus sign, as we did in Fig. B.13.

Number Systems

Objectives

In this appendix, you'll:

- Learn number systems concepts, such as base, positional value and symbol value.
- Understand positional notations in number systems.
- Work with numbers represented in the binary, octal and hexadecimal number systems.
- Abbreviate binary numbers as octal numbers or, more concisely, as hexadecimal numbers.
- Convert octal numbers and hexadecimal numbers to binary numbers.
- Convert back and forth between decimal numbers and their binary, octal and hexadecimal equivalents.
- Learn binary arithmetic and how negative binary numbers are represented using two's complement notation.
- Understand the relationship between one's complement and two's complement notations.
- Write Java numeric literals for binary, octal and hexadecimal numbers.

Outline

C.1 Introduction

C.2 Abbreviating Binary Numbers as Octal and Hexadecimal Numbers

C.3 Converting Octal and Hexadecimal Numbers to Binary Numbers

C.4 Converting from Binary, Octal or Hexadecimal to Decimal

C.5 Converting from Decimal to Binary, Octal or Hexadecimal

C.6 Negative Binary Numbers: Two's Complement Notation

C.1 Introduction

In this appendix, we introduce the number systems programmers use, especially when working on software projects that require close interaction with machine-level hardware. Projects like this include operating systems, computer networking software, compilers,

database systems and applications requiring high performance.

When we write an integer such as 227 or –63 in a Java program, the number is assumed to be in the **decimal (base 10) number system**. The digits in the decimal number system are 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9. The lowest digit is 0, and the highest is 9—one less than the base of 10. Internally, computers use the **binary (base 2) number system** with only the digits 0 and 1. Its lowest digit is 0, and its highest is 1—one less than the base of 2.

Binary numbers are cumbersome to work with because they tend to be much longer than their decimal equivalents. So, two other number systems—the **octal (base 8) number system** and the **hexadecimal (base 16) number system**—are popular primarily because they make it convenient to abbreviate binary numbers. Hexadecimal numbers are also popular for specifying color codes, such as for web-safe colors,[1] in graphics as well as for abbreviating memory addresses. In Section 7.9, we used Base64 encoding in our discussion of private-key cryptography.

1. "Web Safe Color Chart." Accessed October 28, 2024. https://htmlcolorcodes.com/color-chart/web-safe-color-chart/.

In the octal number system, the digits range from 0 to 7. Because both the binary number system and the octal number system have fewer digits than the decimal number system, their digits are the same as the corresponding digits in decimal.

The hexadecimal number system requires 16 digits. Its lowest digit is 0, and its highest digit's value is equivalent to the decimal value 15—one less than the base of 16. By convention, the letters A through F represent the hexadecimal digits corresponding to decimal values 10 through 15. The hexadecimal letters A–F are case insensitive—we'll use upper-case in this appendix. Thus, in hexadecimal, we can have numbers like

- 876 consisting solely of decimal-like digits,
- 8A55F consisting of digits and letters and
- FFE solely composed of letters in the range A–F.

Occasionally, a hexadecimal number spells a common word such as FACE or FEED, which can appear strange to programmers accustomed to working with decimal numbers. The following table contains the digits of the binary, octal, decimal and hexadecimal number systems:

Binary digit
0
1

Octal digit
0
1
2
3
4
5
6
7

Decimal digit
0
1
2
3
4
5
6
7
8
9

Hexadecimal digit
A or a (decimal value 10)
B or b (decimal value 11)
C or c (decimal value 12)
D or d (decimal value 13)
E or e (decimal value 14)
F or f (decimal value 15)

The following table lists the base, lowest digit and highest digit in these number systems:

Attribute	Binary	Octal	Decimal	Hexadecimal
Base	2	8	10	16
Lowest digit	0	0	0	0
Highest digit	1	7	9	F

Each number system uses **positional notation**—each position in which a digit is written has a different **positional value**. For example, in the decimal number 937 (the 9, the 3 and the 7 are referred to as **symbol values**), we say that the 7 is written in the ones position, the 3 is written in the tens position, and the 9 is written in the hundreds position. Each of these positions is a power of the base (10). The powers begin at 0 and increase by 1 as we move from right to left in the number, as in the following table:

Positional values in the decimal number system

Decimal digit	9	3	7
Position name	Hundreds	Tens	Ones
Positional value	100	10	1
Positional val	10^2	10^1	10^0

For decimal numbers with more digits, the next positions to the left would be

- the thousands position (10 to the 3rd power),
- the ten-thousands position (10 to the 4th power),
- the hundred-thousands position (10 to the 5th power),
- the millions position (10 to the 6th power),
- the ten-millions position (10 to the 7th power) and so on.

In the binary number 101, the rightmost 1 is written in the ones position, the 0 in the twos position, and the leftmost 1 in the fours position. Each position is a power of the base (2), and these powers begin at 0 and increase by 1 as we move right to left in the number, as in the table below—so, $101 = 1 * 2^2 + 0 * 2^1 + 1 * 2^0 = 4 + 0 + 1 = 5$.

Positional values in the binary number system

Binary digit			
Position name			
Positional value			
Positional value			

For binary numbers with more digits, the next positions to the left would be

- the eights position (2 to the 3rd power),
- the sixteens position (2 to the 4th power),
- the thirty-twos position (2 to the 5th power),
- the sixty-fours position (2 to the 6th power) and so on.

In the octal number 425, the 5 is written in the ones position, the 2 in the eights position, and the 4 in the sixty-fours position. Each position is a power of the base (8) beginning at 0 and increasing by 1 from right to left in the number, as in the following table:

Positional values in the octal number system

Octal digit			
Position name			
Positional value			
Positional value			

For octal numbers with more digits, the next positions to the left would be

- the five-hundred-and-twelves position (8 to the 3rd power),
- the four-thousand-and-ninety-sixes position (8 to the 4th power),
- the thirty-two-thousand-seven-hundred-and-sixty-eights position (8 to the 5th power) and so on.

In the hexadecimal number 3DA, we say that the A is written in the ones position, the D is written in the sixteens position and the 3 is written in the two-hundred-and-fifty-sixes position. Each position is a power of the base (16) beginning at 0 and increasing by 1 from right to left in the number, as in the following table:

Positional values in the hexadecimal number system

Hexadecimal di			
Position			
Positional value			
Positional value			

For hexadecimal numbers with more digits, the next positions to the left would be

- the four-thousand-and-ninety-sixes position (16 to the 3rd power),
- the sixty-five-thousand-five-hundred-and-thirty-sixes position (16 to the 4th power) and so on.

C.2 Abbreviating Binary Numbers as Octal and Hexadecimal Numbers

Octal and hexadecimal numbers are primarily used to abbreviate lengthy binary representations. The following table highlights the fact that lengthy binary numbers can be expressed concisely in number systems with higher bases than the binary number system:

0	
1	
2	10
3	11
4	100

```
5 101
6 110
7 111
8 1000
9 1001
A 1010
B 1011
C 1100
D 1101
E 1110
F 1111
10 10000
```

An important relationship that both the octal number system and the hexadecimal number system have to the binary system is that their bases (8 and 16 respectively) are powers of the binary number system's base (2). Consider the following 12-digit binary number and its octal and hexadecimal equivalents. Let's determine how this relationship makes it convenient to abbreviate binary numbers in octal or hexadecimal:

Click here to view code image

```
Binary number Octal equivalent Hexadecimal equivalent
100011010001  4321               8D1
```

The binary number converts easily to octal by breaking the 12-digit binary number into groups of three consecutive bits each and writing those groups over the corresponding digits of the octal number as follows:

```
100     011     010     001
4       3       2       1
```

The octal digit under each group of three bits corresponds precisely to the octal equivalent of that 3-digit binary number in the table at the beginning of this section.

A similar relationship can be observed when converting from binary to hexadecimal. Break the 12-digit binary number into groups of four consecutive bits each and write those groups over the corresponding digits of the hexadecimal number as follows:

```
1000    1101    0001
8       D       1
```

The hexadecimal digit under each group of four bits corresponds precisely to the hexadecimal equivalent of that 4-digit binary number in the table at the beginning of this section.

C.3 Converting Octal and Hexadecimal Numbers to Binary Numbers

In the previous section, we saw how to convert binary numbers to their octal and hexadecimal equivalents by forming groups of binary digits and simply rewriting them as their equivalent octal or hexadecimal digits. This process may be applied in reverse to produce the binary equivalent of a given octal or hexadecimal number.

For example, the octal number 653 is converted to binary simply by writing the 6 as its 3-digit binary equivalent 110, the 5 as its 3-digit binary equivalent 101 and the 3 as its 3-digit binary equivalent 011 to form the 9-digit binary number 110101011.

The hexadecimal number FAD5 is converted to binary simply by writing the F as its 4-digit binary equivalent 1111, the A as its 4-digit binary equivalent 1010, the D as its 4-digit binary equivalent 1101 and the 5 as its 4-digit binary equivalent 0101 to form the 16-digit 1111101011010101.

C.4 Converting from Binary, Octal or Hexadecimal to Decimal

We're accustomed to working in decimal, so it's often convenient to convert a binary, octal, or hexadecimal number to decimal to get a sense of what the number is "really" worth. Our diagrams in Section C.1 express the positional values in decimal. To convert a number to decimal from another base, multiply the decimal equivalent of each digit by its positional value and sum these products. For example, the binary number 110101 is converted to decimal 53, as shown in the following table:

Converting a binary number to decimal

Symbol						
Position						
Product						
Sum:						

```
32 + 16 + 0 + 4 + 0 + 1 = 53
```

To convert octal 7614 to decimal 3980, we use the same technique, this time using appropriate octal positional values, as shown in the following table:

Converting an octal number to decimal

Symbol	7	6	1	4
Positional value	512	64	8	1
Products	3584	384	8	4

Sum: 3584 + 384 + 8 + 4 = 3980

To convert hexadecimal AD3B to decimal 44347, we use the same technique, this time using appropriate hexadecimal positional values, as shown in the following table:

Converting a hexadecimal number to decimal

Symbol	A (10 decimal)	D (13 decimal)	3	B (11 decimal)
Positional value	4096	256	16	1
Products	40960	3328	48	11

Sum: 40960 + 3328 + 48 + 11 = 44347

Java Numeric Literals for Binary, Octal and Hexadecimal Numbers

Java supports binary, octal and hexadecimal literal values:

- A binary value like 110101 (decimal 53) can be written as a binary literal in Java code with a leading `0b` as `0b110101`. You can separate the digits in any numeric literal with underscores (_) for readability, as in `0b110_101`.
- An octal value like 7614 (decimal 3980) can be written as an octal literal in Java code with a leading `0` as `07614`.
- A hexadecimal value like AD3B (decimal 44347) can be written as a hexadecimal literal in Java code with a leading `0x` or `0X` as `0xAD3B` or `0XAD3B`.

Java converts binary, octal and hexadecimal literals to their decimal equivalents. To confirm this, you can use `System.out`'s printing methods to output the preceding list's octal, binary and hexadecimal literal values. In addition, each integer type-wrapper

class has `static` methods that can convert integer values to binary, octal or hexadecimal `String`s.

C.5 Converting from Decimal to Binary, Octal or Hexadecimal

The conversions in Section C.4 follow naturally from the positional-notation conventions. Converting from decimal to binary, octal, or hexadecimal also follows these conventions.

Converting Decimal to Binary

Suppose we wish to convert decimal 57 to binary. We begin by writing the decimal positional values of the columns from right to left until we reach the first column whose positional value is greater than the decimal number. We don't need that column, so we discard it. Thus, we first write:

Click here to view code image

```
Positional values: 64    32    16    8    4    2    1
```

Then we discard the column with positional value 64, leaving:

Click here to view code image

```
Positional values: 32    16    8    4    2    1
```

Next, we work from the leftmost column to the right. We divide 32 into 57 and observe that there's one 32 in 57 with a remainder of 25, so we write 1 in the 32 column. Next, we divide 16 into 25 and observe that there's one 16 in 25 with a remainder of 9 and write 1 in the 16 column. We then divide 8 into 9 and observe that there's one 8 in 9 with a remainder of 1. The next two columns each produce quotients of 0 when their positional values are divided into 1, so we write 0s in the 4 and 2 columns. Finally, 1 into 1 is 1, so we write 1 in the 1 column. This yields:

Click here to view code image

```
Positional values: 32    16    8    4    2    1
Symbol values:     1     1     1    0    0    1
```

Thus, decimal 57 is equivalent to binary 111001.

Converting Decimal to Octal

To convert decimal 103 to octal, we begin by writing the decimal positional values of the columns from right to left until we reach the first column whose positional value is greater than the decimal number. We do not need that column, so we discard it. Thus, we first write:

Click here to view code image

```
Positional values: 512    64    8    1
```

Then we discard the column with positional value 512, yielding:

Click here to view code image

```
Positional values:       64    8    1
```

Next, we work from the leftmost column to the right. We divide 64 into 103 and observe that there's one 64 in 103 with a remainder of 39, so we write 1 in the 64 column. We divide 8 into 39 and observe that there are four 8s in 39 with a remainder of 7 and write 4 in the 8 column. Finally, we divide 1 into 7 and observe that there are seven 1s in 7 with no remainder, so we write 7 in the 1 column. This yields:

Click here to view code image

```
Positional values:  64    8    1
Symbol values:      1     4    7
```

and thus decimal 103 is equivalent to octal 147.

Converting Hexadecimal to Binary

To convert decimal 375 to hexadecimal, we begin by writing the decimal positional values of the columns from right to left until we reach a column whose positional value is greater than the decimal number. We do not need that column, so we discard it. Thus, we first write:

Click here to view code image

```
Positional values:  4096    256    16    1
```

Then we discard the column with positional value 4096, yielding:

Click here to view code image

```
Positional values:      256     16     1
```

Next, we work from the leftmost column to the right. We divide 256 into 375 and observe that there's one 256 in 375 with a remainder of 119, so we write 1 in the 256 column. We divide 16 into 119 and observe that there are seven 16s in 119 with a remainder of 7 and write 7 in the 16 column. Finally, we divide 1 into 7 and observe that there are seven 1s in 7 with no remainder, so we write 7 in the 1 column. This yields:

Click here to view code image

```
Positional values:  256     16     1
Symbol values:      1       7      7
```

and thus decimal 375 is equivalent to hexadecimal 177.

C.6 Negative Binary Numbers: Two's Complement Notation

The discussion so far in this appendix has focused on positive numbers. This section explains how computers represent negative binary numbers using **two's complement notation**. First, we explain how to form the two's complement of a binary number, then we show why it represents the negative value of the given binary number.

Java uses 32-bit `int`s. Suppose

```
int value = 13;
```

The 32-bit representation of `value` is

Click here to view code image

```
00000000 00000000 00000000 00001101
```

To form the negative of `value`, we first form its **one's complement** by applying Java's bitwise complement operator (`~`):[2]

2. Visit `https://deitel.com/javafp5` for an intro to the Java bitwise operators.

Click here to view code image

```
onesComplementOfValue = ~value;
```

Internally, `~value` is now `value` with each of its bits reversed—ones become zeros and zeros become ones, as follows:

Click here to view code image

```
value:
00000000 00000000 00000000 00001101

~value (i.e., value's one's complement):
11111111 11111111 11111111 11110010
```

To form the two's complement of `value`, add 1 to `value`'s one's complement. Thus

Click here to view code image

```
Two's complement of value:
11111111 11111111 11111111 11110011
```

If this is in fact equal to –13, we should be able to add it to binary 13 and obtain a result of 0. Let's try this:

Click here to view code image

```
 00000000 00000000 00000000 00001101
+11111111 11111111 11111111 11110011
-----------------------------------
 00000000 00000000 00000000 00000000
```

The `1` **carry bit** coming out of the leftmost column is discarded, and we indeed get 0 as a result. If we add the one's complement of a number to the number, the result would be all 1s. The key to getting a result of all zeros is that the two's complement is one more than the one's complement. Adding 1 causes each column's sum to be 0 with a carry of 1. The carry keeps moving left until it's discarded from the leftmost bit. Thus, the resulting number is all zeros.

Computers actually perform a subtraction, such as

```
x = a - value;
```

by adding the two's complement of `value` to `a`, as follows:

```
x = a + (~value + 1);
```

Suppose `a` is 27 and `value` is 13 as before. If the two's complement of `value` is actually the negative of `value`, then adding the two's complement of `value` to `a` should produce the result 14. Let's try this:

Click here to view code image

```
a (i.e., 27)       00000000 00000000 00000000 00011011
+(~value + 1)     +11111111 11111111 11111111 11110011
                  -----------------------------------
                  00000000 00000000 00000000 00001110
```

which is indeed the binary equivalent of 14.

Index

[*Note:* Page references for defining occurrences of terms appear in **bold**.]

Symbols

_ (underscore) as a variable name 701
_ digit separator **58**, 139
_ identifier 50
_ SQL wildcard character **837**, 837
, (comma) formatting flag **108**
! logical NOT 119, **121**
!= not equals 62
? (wildcard type argument) **515**
?: ternary conditional operator **77**
. (current folder in a path) 409
.. (enclosing folder in a path) 409
, flag **1009**
"omni-moderation-latest" model **794**
(flag **1009**
[] (regex character class) **243**
{ left brace 51
{*n*,} (quantifier in regex) **245**
{*n*,*m*} (quantifier in regex) **245**
} right brace 51
@Override annotation **325**
* (quantifier in regex) **243**
* multiplication 60, 62

* SQL wildcard character **834**, 834
* wildcard in a filename **268**
*= multiplication assignment operator 91
/ division 60, 62
/ forward slash in end tags **647**
/! JShell command **953**, 978
/? JShell command **978**
/* */ traditional comment **49**
/** */ Java documentation comment 49
// end-of-line comment **49**
/= (division assignment operator) 91
/drop JShell command **980**
/edit JShell command 979
/help JShell command **978**
/id JShell command **952**, 978
/reload JShell command **979**
/set feedback JShell command **980**
/set JShell command 982
/set start JShell command 979, **982**
/vars JShell command 978
\ (regex metacharacter) **242**
\ backslash escape sequence 55
\', single-quote-character escape sequence 1013
\", double-quote escape sequence 55, 1013
\\, backslash-character escape sequence 1013
\b, escape sequence 1013
\D (regex character class) **242**
\d (regex character class) 242, **243**
\f, form-feed escape sequence 1013
\n, newline escape sequence 55, 1013
\r, carriage-return escape sequence 1013
\S (regex character class) **242**
\s (regex character class) **242**
\t, horizontal tab escape sequence 55, 1013
\W (regex character class) **242**
\w (regex character class) **242**
& boolean logical AND 118, **121**

&& conditional AND **119**, 120
# flag **1009**, 1011
% conversion character **1006**
% remainder 60, 62
% SQL wildcard character **837**, 837
%% format specifier 1007
%= (remainder assignment operator) 91
%A format specifier 1004
%a format specifier 1004
%B format specifier **1006**
%b format specifier **67**, 122, 1006
%C format specifier **1005**
%c format specifier 1005
%d format specifier **59**, 1002
%E format specifier 1004, **1004**
%e format specifier 1004, **1004**
%f format specifier **86**, 151, 1004
%G format specifier **1004**
%g format specifier **1004**, 1004
%H format specifier **1006**
%h format specifier 1006
%n format specifier 1006
%o format specifier **1003**, 1004
%S format specifier **1005**
%s format specifier **56**, 1002, 1005
%X format specifier **1003**
%x format specifier **1003**
^ (regex metacharacter) **243**
^ boolean logical exclusive OR 119, **121**
+ (quantifier in regex) **244**
+ addition 60, 62
– flag **1009**
+ flag **1009**
– (minus sign) formatting flag **107**
– subtraction 60, 62
–– predecrement/postdecrement **91**
++ preincrement/postincrement **91**

+= addition assignment operator **91**
< less than 62
<!--…--> XML comment tags 646
<= less than or equal 62
<> angle brackets for XML elements **646**
<> diamond notation for generic type inference 463
-= (subtraction assignment operator) 91
-=(subtraction assignment operator) 91
== is equal to 62
== to determine whether two references refer to the same object 332
-> (arrow token in a lambda) **526**
> greater than 62
>= greater than or equal to 62
| boolean logical inclusive OR 119, **121**
|| conditional OR 118, **120**
~, bitwise complement operator 1025

Numerics

0 format flag **179**, 284
0x (hexadecimal prefix) 1011
12-hour clock format 163
24-hour clock format 163

A

abbreviating assignment expressions 91
abs method of Math 153
absolute path **409**, 410
absolute positioning 599
absolute value 153
abstract class 337
abstract class **336**, 337, 338

    Employee 338

abstract keyword **337**

`abstract` method 50, **337**, 340
abstract superclass **336**
Abstract Window Toolkit (AWT) 568
access modifier **264**, 265

- `private` 286, 370
- `protected` 286, **370**
- `public` **264**, 286, 370

access modifier in the UML - (private) 268
accessibility xxxviii, 632, 890

- image description 771, 802

accessible image descriptions 802
accessor method **296**
`Accordion` class (JavaFX) 599
accumulation function associative 528
action expression in the UML **71**
action state symbol in the UML **71**
`ActionEvent` class (JavaFX) 591, 592
activation record **156**
activity diagram (in the UML) **71**, 71, 74, 78
activity in the UML **71**
actual type arguments **499**
`add` method

- `ArrayList<E>` **203**
- `BigDecimal` **127**
- `BigInteger` **97**, 925
- `LinkedList<T>` **468**
- `List<T>` **460**, 463

`--add-modules` option of `javac` 578
add packages to JShell's `CLASSPATH` 975
add rows or columns to a `GridPane` 583
`addAll` method

- `Collections` 469, **478**
- `List` **464**

`addFirst` method
    of `ArrayList<E>` 200
    of `LinkedList` **468**
adding `String`s **67**
addition 60, 61
addition compound assignment operator `+=` **91**
`addLast` method
    of `ArrayList<E>` 200, 507
    of `LinkedList` **468**
`addListener` method of interface `ObservableValue` **593**
Adobe Firefly image generator 32
Adobe Firefly video generator 32, 821
Advanced Encryption Standard (AES) 253, **253**
    256 bit keys 257
    key size **257**
advanced reasoning models 779
AES encryption algorithm xli, 258
agent in AI 45
aggregator module **891**
    `java.se` **891**
agile software development **26**
AI (artificial intelligence) xxvii, xlix
AI agent 45
AI chatbot 36
`Alert` class **852**, 863
`AlertType.ERROR` 852
algebraic notation 61
algorithm 926
    in Java Collections Framework 468
aligning decimal points in output 1002
alignment in a `VBox` (JavaFX) **573**

`Alignment` property 573

`Alignment` property of a `VBox` (JavaFX) 574
alphabetizing text 212
AlphaGo 34
alphanumeric character **242**
AlphaZero **34**
Amazon Alexa xliv, 239
Amazon Web Services (AWS) 20
`AmbientLight` class 691
`AnchorPane` layout (JavaFX) 599, 649
`and` method

of functional interface `IntPredicate` **553**
of functional interface `Predicate` **553**

AndroidXR 23
angle brackets (`<` and `>`)

for generics **499**
for XML elements **646**

animation xlii

cycle indefinitely 675
frame-by-frame 677

animations and sprites 694
`AnimationTimer` class **677**

`handle` method **678**

annotation 428

`@Override` **325**
compile-time 428
runtime 428

anonymous inner class 263, **424**, 679
anonymous method (lambda) **526**
Anthropic 759
Anthropic Claude lxvii, 36, 239

API (application programming interface) 21, 57, 154
API-based generative AI applications lii, 758
`apiKey` method of `SimpleOpenAI-Builder` **799**
`append` method of class `String-Builder` **228**
Apple
    Siri xliv
Apple Siri 239
`Application` class (JavaFX) **579**, 605, 613
    `launch` method **579**, 589
    `start` method **579**, 589
application programming case studies xlii
application programming interface (API) **11**, 154
AR (augmented reality) **22**
`Arc` class (JavaFX) **649**, 651
    `length` property **651**
    `radiusX` property **651**
    `radiusY` property **651**
    `startAngle` property **651**
    `type` property **651**
archive 881
`ArcTo` element of a `Path` **658**
`ArcType enum` **689**
`args` parameter 194
argument index 1002, **1012**
argument list 1003
argument promotion **159**
argument to a method **52**, 67, 135
arithmetic calculation 60
arithmetic compound assignment operators **91**
arithmetic operators 60
arithmetic overflow **82**, 377
`ArithmeticException` class **379**, 384
ARPANET 18
array **170**

bounds checking **183**
ignoring element zero 183
jagged 190
`length` instance variable **172**
pass an array element to a method 186
pass an array to a method 186

array-access expression **172**
array-creation expression **172**
array initializer **175**

for multidimensional array 190
nested **190**

array of one-dimensional arrays 190
`ArrayBlockingQueue` class **734**, 740

`put` method **734**, 736
`size` method **736**
`take` method **734**, 736

`arraycopy` method of class `System` **197**, 199
`ArrayIndexOutOfBoundsException` class **183**, 184
`ArrayList<E>` generic class xlix, **199**, 200, 459

`add` method **203**
`addFirst` method 200
`addLast` method 200
`clear` method 200, **203**
`contains` method 200, **203**
`get` method **203**
`getFirst` method 200, **442**
`getLast` method 200, **442**
`indexOf` method 200, **203**
`isEmpty` method 297
`remove` method 200, **203**
`removeFirst` method 200
`removeLast` method 200, 507
`size` method **203**

`Arrays` class **197**

`asList` method **466**, 467, 543
`equals` method **197**, 199
`fill` method **197**
`parallelSetAll` method **701**
`parallelSort` method 199, 701
`sort` method **197**, 198, 539, 701
`stream` method **543**, 543
`toString` method **197**, 198

arrow token (`->`) in a lambda **526**
arrow (`->`) symbol in a `switch` expression **141**, 146, 147
arrow in the UML 71
artificial general intelligence (AGI) **33**
artificial intelligence (AI) xxvii, xlix, 32, 240, **248**, 407
artificial neural network **35**
ascending order 198

`ASC` in SQL 838

`asList` method of `Arrays` **466**, 543
`assert` statement 50, **398** assertion **398**
`AssertionError` class **398**
Assigning superclass and subclass references to superclass and subclass variables 334
assignment operator 61, **91**

`*=` 91
`/=` 91
`%=` 91
`-=` 91
`=` **60**

`"assistant" role` in an OpenAI chat completion **767**
assisting people with disabilities 31
associative accumulation function 528
associative operation in a reduction 537

`asString` method of class `DoubleProperty` **616**

asterisk (`*`) SQL wildcard character **834**
asymmetric-key cryptography **253**, 446
asynchronous 797
asynchronous event **377**
atomic operation 864
attribute
    in the UML 268
    of an object 8, 10
attribute in XML 646
audio 783
    `"whisper-1"` model **783**
Audio API (OpenAI) **784**
audio recording 786
audio segment in a transcription 811
`AudioClip` class (JavaFX) **662**
`Audios` class **806**
    `speak` method **807**
    `transcribe` method **806**
`audios` method of `SimpleOpenAI` **806**
augmented reality (AR) xlii, **22**, 23
`author_ISBN` table of `books` database 833, 834
`authors` table of `books` database 833
auto commit state 865
autoboxing 236, 504
    an `int` 504
`AutoCloseable` interface 357, **400**, 707, 843
    `close` method 400
auto-completion **947**, 964, 973
autoincremented value **833**, 833, 840
automated
    closed captioning 31

automatic driver discovery (JDBC 4) **843**
automatic garbage collection 388
automatic module **895**
automobile as an object 9
average 79, 81, 83, 441, **442**
`average` method of interface `DoubleStream` 557
`average` method of interface `Int-Stream` **536**
`awaitTermination` method of class `ExecutorService` 715
`awaitTermination` method of interface `ExecutorService` **710**, 715
AWT (Abstract Window Toolkit) 568

B

`B` conversion character **1006**
`b` conversion character **1006**
backslash (`\`) **55**
backtracking **941**
bad data **438**
bandwidth 19
banker's rounding **127**
bar chart 177, 179
bar of asterisks 177, 179
base case **920**, 925, 926, 931
base of a number 235
Base64 **255**, 803

- alphabet **255**
- encode a `byte` array 258
- encoded text 258

`Base64` class

- `decode` method **258**
- `encodeToString` method **255**
- `getDecoder` method **258**
- `getEncoder` method **255**

`Base64Util` class

`encode` method **803**

behavior

of a class 8

`between` method of class `Duration` **164**, **702**
`between` method of `Period` class **164**
bias (generative AI) 45
`BiConsumer` functional interface **556**, 562
bidirectional iterator **465**
bidirectional property binding 616
big data xlii, xlix, **28**, 30, 407, 432
Big *O* 943

notation **942**

Big O notation **920**
`BigDecimal` class 86, 109, **124**, 125, 590

`add` method **127**
`multiply` method **127**
`ONE` constant **127**
`pow` method **127**
rounding policy **125**, 125
`setScale` method **127**
significant digits 125
`TEN` constant **127**
`TWO` constant **127**
`ZERO` constant **127**

`BigInteger` class **95**, 742

`add` method **97**, 925
`multiply` method **97**
`ONE` constant 925
`subtract` method **97**, 925
`valueOf` method **97**
`ZERO` constant 925

binary

base 2 number system 1018

binary file **408**
binary literal value 1023
binary operator **59**, 60, 121
binary search 15

algorithm 477
`Arrays` class `binarySearch` method **197**
`binarySearch` method of `Arrays` 199

binary search algorithm 477
`BinaryOperator` functional interface 540
`binarySearch` method of `Collections` 468, **477**, 478
bind a collection to a `ListView` 598
`bind` method of class `StringProperty` **616**
`Bindings` class 668

`selectDouble` method **668**
`when` method **668**

Bitcoin 7, 24, 30, 95, 259
bitwise complement operator (`~`) 1025
bitwise operators 119
blank line 50
block **64**, 75
Block Increment property of a `Slider` (JavaFX) 587
blockchain xlii, 7, **24**, 30, 95, 259

blocks 24

`BlockingQueue` interface **734**

`put` method **734**, 736
`take` method **734**, 736

blocks in a blockchain 24
blog, `https://deitel.com/blog` xli
body

class declaration **51**

of a method **51**

Bohm, C. 71
`books` database 832
`Boolean` class **458**
`boolean`

promotions 160

boolean logical AND `&` 118, **121**
boolean logical exclusive OR `^` 119, **121**
boolean logical inclusive OR `|` **121**
`boolean` primitive type 50, **73**
Boolean values in JSON 419
`BorderPane` class (JavaFX) 598, 599, **601**
bound instance method reference **533**
bounded buffer **739**
bounding box 688, 689
bounds checking **183**
`Box` class **690**
`boxed` method of interface `IntStream` **562**
boxing a primitive value into an object 200
boxing conversion **459**, 504
braces (`{` and `}`) 64, 75, 175

not required 114

brain mapping 31
`break` statement 50, **114**, 116
brittle software **371**
brute force computing 33
buffer 733
`build` method of `ChatRequest-Builder` **800**
`build` method of `HttpRequest.Builder` **430**
`build` method of `SimpleOpenAI-Builder` **799**
`build` method of `TranscriptionRequestBuilder` **806**
build tools 888
builder design pattern **716**, 797
`builder` method

`ChatRequest` **800**

`builder` method of `ImageRequest` **809**
`builder` method of `ModerationRe-quest` **811**
`builder` method of `SpeechRequest` **807**
`builder` method of `Transcription-Request` **806**
`Button` class (JavaFX) **570**, 580, 585

Max Width property 604
Text property 604

`ButtonBar` class (JavaFX) 599
byte-based stream **407**
`Byte` class **458**
`byte` primitive type 50, 111

promotions 160

bytecode **13**, 53
bytecode verifier **14**

C

cached thread pool **709**, 709, 710, 718
Caesar cipher 252 calculations 71
`call` method of class `Task` **742**, 743
`call` method of interface `Callable` **717**
`Callable` interface **717**

`call` method **717**
cancel execution 717
determine if canceled 717
determine if completed 717

`CallableStatement` interface **864**
`CallBack` interface (JavaFX) **624**
callback method 797
calling method (caller) 134
camel case **51**

`cancel` method of class `SwingWorker` **753**

`cancel` method of class `Task` **753**
canceled state of a `Task` 746
cancer diagnosis 31
`Canvas` class (JavaFX) 685

`getGraphicsContext2D` method **686**

capacity of a `StringBuilder` 225
`capacity` method

of class `StringBuilder` **226**

capturing lambdas **542**
card games 278
Card Shuffling and Dealing with `Collections` method `shuffle` 473
caret (`^`) regex metacharacter **243**
carrier for a virtual thread **714**
carry bit 1026
Cascading Style Sheets (CSS) **569**, 569, 633, 640, 641

`image-pattern` function **654**, 688
`linear-gradient` function **654**, 689
`radial-gradient` function **653**
`rgba` function **653**
type selector **651**

case folding 250
case insensitive 247

regular expression 241

`case` keyword 50, **114**
case sensitive **51**

regular expression 241

`CASE_INSENSITIVE` constant of class

`Pattern` **247**

case-sensitive 247
casino 142
cast operator **85**, 160
    downcast **346**
    inserted during type erasure 504
`catch`
    a superclass exception 385
    an exception 380
    block **183**, 382, 384, 390, 392
    clause **382**
    keyword 50, 382
    multi-`catch` **382**
catch-or-declare requirement **387**
`cd` to change directories 52
`ceil` method of `Math` 153
cell in a `GridPane` 580
`centerX` property of a `Circle` **651**
`centerX` property of a `Ellipse` **651**
`centerY` property of a `Circle` **651**
`centerY` property of an `Ellipse` **651**
chain of constructor calls 329
chained exception **394**
`changed` method of interface `Change-Listener` (JavaFX) **594**
    `ChangeListener` interface (JavaFX) **581**, 593
    `changed` method **594**
`char`
    array 211
    keyword 50
    primitive type **58**, 111
    promotions 160
character
    constant **116**

literal **209**
string **51**

character-based stream **407**
`Character` class 208, 234, 458

`charValue` method **236**
`digit` method **235**
`forDigit` method **235**
`isDefined` method **234**
`isDigit` method **234**
`isJavaIdentifierPart` method **234**
`isJavaIdentifierStart` method **234**
`isLetter` method **234**
`isLetterOrDigit` method **234**
`isLowerCase` method **234**
`isUpperCase` method **234**
`static` conversion methods 235
`toLowerCase` method **235**
`toUpperCase` method **235**

character class (regular expressions) **242**, 243

custom **243**

`charAt` method

of class `String` 211
of class `StringBuilder` **227**

`charValue` method of `Character` **236**
`Chat` class `firstContent` method **800**
chat completion xxxvii, **764**, 766
chatbots 32, 36, **239**

Anthropic Claude lxvii
Google Gemini lxvii
OpenAI ChatGPT lxvii
Perplexity lxvii

`ChatCompletions` class **800**

`create` method **800**

`chatCompletions` method of `SimpleOpenAI` **800**
ChatGPT xliv, 36

Advanced Voice Mode 759
AI chatbot 760

`ChatMessage` class **800**
`ChatRequest` class

`builder` method **800**

`ChatRequestBuilder` class **800**

`build` method **800**
`message` method **800**
`model` method **800**
`responseFormat` method **805**

checked exception **386**, 387, 397
chess 33
`children` element of a `VBox` **647**
`ChronoUnit enum` 162

`DAYS` constant **165**

cipher

Caesar 252
substitution 252

`Cipher` class

`DECRYPT_MODE` constant **258**
`doFinal` method **258**, 259
`ENCRYPT_MODE` constant **258**
`getInstance` method **258**, 258

ciphertext in cryptography **253**
`Circle` class (JavaFX) **601**, 608, 649

`centerX` property **651**

`centerY` property **651**
`radius` property **651**

class **9**

anonymous inner class 263
camel case naming 134, 264
cannot extend a final class 348
`class` keyword 264
constructor **211**, 267, 271
declaration **50**
declaration body **51**
default constructor **273**
file **53**
fully qualified name 270
hierarchy **321**, 337
instance creation expression **210**, 267
instance creation expression diamond notation (<>) **203**
instance variable **10**
library 319, **319**
literal 909
loader **14**
method **152**
name **50**
nested 801
nested class 263
`sealed` **367**

class-average problem 79, 80, 84
`Class` class 332, **346**

`getName` method 332, **346**

`.class` file **13**, 53

separate one for every class 288

`class` keyword **50**, 50, 264
class name of a CSS class selector 642

class variable **153**, 303
classwide information **303**
`Class<T>` class 804
`getSimpleName` method **805**
Classes
`Accordion` (JavaFX) 599
`ActionEvent` (JavaFX) 591, 592
`Alert` **852**, 863
`AmbientLight` 691
`AnchorPane` (JavaFX) 599
`AnimationTimer` **677**
`Application` (JavaFX) **579**, 605, 613
`Arc` 649, 651
`ArithmeticException` **379**
`ArrayBlockingQueue` **734**, 740
`ArrayIndexOutOfBoundsException` 183, 184
`ArrayList<E>` **199**, 200, 203
`ArrayList<T>` 459, 460
`Arrays` **197**
`AssertionError` **398**
`AudioClip` 662
`BigDecimal` 86, 109, **124**, 125, 590
`BigInteger` 95, 742
`Bindings` 668
`Boolean` **458**
`BorderPane` (JavaFX) 598, 599, **601**
`Box` **690**
`ButtonBar` (JavaFX) 599
`Byte` **458**
`Canvas` 649, 685
`Character` 208, 228, 234, 458
`ChatMessage` **800**
`Circle` 649, 651
`Circle` (JavaFX) 608
`Circle` class (JavaFX) **601**

Class 332, **346**
Class<T> 804
Collections **458**
Collector **534**, 544
Color **939**
ColorPicker 935, **935**
ConcurrentHashMap 740
ConcurrentModificationException 462
CsvMapper **434**, 436
CsvSchema **434**, 436
Cylinder **690**
DeveloperMessage **800**
DialogPane (JavaFX) 599
Directional 691
DirectoryChooser **625**, 629
Double **458**, 513
DoubleProperty (JavaFX) 593, **616**
DriverManager **844**
Ellipse 649, 651
EnumSet **303**
Error **385**
Exception **384**
Executors **707**
FadeTransition **673**
FileChooser **625**, 627
Files **409**, 558
FillTransition **672**
FindException 904, 905
Float **458**
FlowPane (JavaFX) 599
Formatter **408**, 1002, 1014
FXCollections (JavaFX) **617**
FXMLLoader (JavaFX) **581**, 589, 592
GraphicsContext **685**
GridPane (JavaFX) **579**, 579, 580, 599
HashMap **486**
HashSet **483**

HBox (JavaFX) 599
HttpClient 430
Image (JavaFX) **621**
ImageView (JavaFX) **571**
InputMismatchException **379**
Instant **702**
Integer **458**, 513
Interpolator **673**
InterruptedException **708**
IntSummaryStatistics **536**
KeyFrame **675**
KeyPair **447**
KeyPairGenerator **447**
Label 641
Label (JavaFX) **571**
Line 649, 651
LinkedBlockingQueue 740
LinkedList **459**
ListCell (JavaFX) **622**
ListView (JavaFX) 598
Long **97**, 458
MappingIterator **436**
Matcher 209, **246**
Math 152, 153
Media 662, 665, 666
MediaPlayer 662, 665, 667, 668
MediaView 662, 665, 667
MouseEvent (JavaFX) **608**
Node (JavaFX) **570**, 602
Number **458**, 513
NumberFormat **580**, 590, 703
ObjectMapper **420**
ObjectWriter **434**
ObservableList(JavaFX) **608**, 846, **846**
Optional class 551
OptionalDouble **536**, 557
OptionalInt **536**

Paint **686**, 687, 688, 689
Pane (JavaFX) 598, 599, **602**, 649
ParallelTransition **672**
Parent (JavaFX) **581**, 589
Path 655
PathElement 655
PathTransition **673**
Pattern 209, **246**
PhongMaterial **692**
Platform **741**
PointLight 691
Polygon 655
Polyline 655
PriorityQueue **480**
RadioButton (JavaFX) 598
RadioButtonMenuItem (JavaFX) 608
ReadOnlyObjectProperty (JavaFX) **621**, 668
Rectangle (JavaFX) 598, **649**, 651
Rotate 660
RotateTransition **673**
RuntimeException **386**
Scale 660
ScaleTransition **674**
Scanner **58**
Scene (JavaFX) **570**, 579, 589, 590
ScopedValue 724, **726**
ScopedValue.Carrier **726**
ScrollPane (JavaFX) 599
SecretKeySpec **451**
SecureRandom **142**, 257
SelectionModel (JavaFX) **621**
SequentialTransition **674**
ServiceLoader 905, **909**
Shape 649
Shape3D 649, 690
Short **458**
Slider (JavaFX) **570**, 577, 580

Sphere 690
SplitPane (JavaFX) 599
SpotLight 691
SQLException 844
StackTraceElement 393
StackWalker 393, 565
Stage (JavaFX) 570, 579, 589, 590
Stop 688
String 65, 67, 208
StringBinding (JavaFX) 616
StringBuilder 208, 225
StringJoiner 544
StringProperty (JavaFX) 616
StrokeTransition 672
StructuredTaskScope 724, 727
StructuredTaskScope.Shut-downOnFailure 727
SystemMessage 800
Tab (JavaFX) 599
TableView (JavaFX) 631
TabPane (JavaFX) 599
Task 741, 742
TextArea 625
TextField (JavaFX) 570, 580
TextFlow (JavaFX) 600
Throwable 385, 392
TilePane (JavaFX) 600
Timeline 675
TitledPane (JavaFX) 598, 600, 601, 604
ToggleButton (JavaFX) 608
ToolBar (JavaFX) 600, 625, 935
Transform 660
Translate 660
TreeMap 486
TreeSet 483
TreeTableView (JavaFX) 631
Types 845
UnsupportedOperationException 467

VBox (JavaFX) **573**, 600, 641

classpath **422**, 843, 884

cp option **422**

CLASSPATH environment variable 884
Claude chatbot 36
cleaning and preparing data for analysis 209, 241
clear method

of ArrayList<E> 200, **203**
of List<T> **466**
of PriorityQueue **480**

clearRect method of class GraphicsContext class

clearRect method **940**

click a button 580
client of a class **270**
client application **424**
clone method of Object 332
cloning objects

deep copy **332**
shallow copy **332**

close a window 580
close method

of Connection 845
of Formatter **415**
of interface Connection 845
of interface ResultSet 845
of interface Statement 845
of ResultSet 845
of Scanner **417**
of Statement 845

close method of class ExecutorService 715
close method of interface Auto-Closeable 400

`close` method of interface `Executor-Service` **710**
closed captions (subtitles) **790**
closed captions in VTT format xxxviii
closed range of integers 523
`ClosePath` element of a `Path` **659**
cloud 20, 418, 424
    computing **20**
cloud services
        Amazon Web Services (AWS) 20
        Google Cloud Platform (GCP) 20
        Microsoft Azure 20
cloud-based services **20**
cloud-based services **418**, 426
code 11
code generation 776
code maintenance 277
code repositories (on GitHub) 11
coding errors (preventable) 387
coin tossing 138
`collect` method
        of interface `IntStream` **534**, **535**
        of interface `Stream` **544**, 555, 556, 562
`collect` method of interface `Stream` 555
collection **170**, 457
    concurrent 700
`Collection` interface **457**, 457, 461, 468
        `contains` method **462**
        `iterator` method **461**
        `removeAll` method **462**
`Collections` class **458**
        `addAll` method 469, **478**

`binarySearch` method 468, **477**, 478
`copy` method 469, **475**
`disjoint` method 469, **479**
`fill` method 469, **475**
`frequency` method 469, **479**
`max` method 469, **475**
`min` method 469, **475**
`reverse` method 468, **475**
`reverseOrder` method **470**
`shuffle` method 469, **472**, 474
`singleton` method 493
`singletonList` method 493
`singletonMap` method 493
`sort` method **469**

collections framework **456**

convenience factory methods **489**
interfaces 457

collector 534
`Collector` interface **544**
`Collectors` class **534**, 544

`groupingBy` method **555**, 556, 559, 562
`joining` method **534**
`toList` method **544**

collision detection 694
collision in a hashtable **482**
`Color` class **939**

`rgb` method 616

color model RGBA **611**
color stops **653**
`ColorPicker` class 935, **935**
column 189

in a database table **831**, 831, 833

columns of a two-dimensional array 189
comma (,) formatting flag **108**
comma in an argument list 56
comma-separated list 103

of arguments **56**
of parameters 134

comma-separated value (CSV) files 419
command line **52**

argument **154**, 194

Command Prompt 52
command window **52**
comment

end-of-line (single-line), `//` **49**, 52
Javadoc **49**
single line 52

comment in XML 646
`CommissionEmployee` class derived from `Employee` 342
commit a transaction **864**
`commit` method of interface `Connection` **865**
`Comparable<T>` interface 356, 469, 502

`compareTo` method 469, **502**

`Comparator` interface **469**, 471, 547

`compare` method **471**
`thenComparing` method **552**

`Comparator` object 470, 475, 484, 486

in `sort` 470

`compare` method of interface `Comparator` **471**

`compareTo` method
of class `String` **215**
of `Comparable` 469

`compareTo` method of `Comparable<T>` **502**
comparing `String` objects 212
comparison operator 356
compartment in a UML class diagram 268
compilation error **50**, 53
    JShell 950
compile 52, 53
compile a module 876
compile a program **13**
`compile` method of class `Pattern` **246**
compile-time type safety 460
compile-time annotation 428
compile-time polymorphism 347
compile-time type checking 457
compile-time type safety **496**
compiling an application with multiple classes 268
`CompletableFuture` interface 797
    `join` method 797
complexity theory 926
component 8
composite primary key **834**, 835
composition xli, **297**, 321
    dependency injection 359
compound assignment operators **91**, 93
compound condition **119**
compound interest 106
computer science xlii
computer vision 31
    applications 32
computerized scientific notation **1004**
`concat` method of class `String` **221**
concrete class **336**
concrete subclass 340

concurrency xlii
Concurrency API 700
concurrent access to a `Collection` by multiple threads 492
concurrent collection xlix, 700
concurrent operations 698
concurrent programming 4, **699**
`ConcurrentHashMap` class 740
`ConcurrentModificationException` class 462
condition 62, 110

    compound **119**

conditional AND `&&` **119**, 121
conditional expression **77**
conditional operator `?:` 77
conditional OR `||` 118, **120**
connect to a database 842
connection between Java program and database 844
`Connection` interface **844**, 845, 865

    `close` method 845
    `commit` method **865**
    `createStatement` method **844**
    `getAutoCommit` method **865**
    `prepareStatement` method **855**
    `rollBack` method **865**
    `setAutoCommit` method **865**

connection string (database URL) 843
`const` keyword 50
constant **116**, 308
constant integral expression **111**, 116
constant run time **942**
constant variable **116**, 176

    must be initialized 176

constructor **211**, 267, 271

    call another constructor of the same class using `this`

290
call chain 329
copy an object's data 291
injection **361**
multiple parameters 274
no-argument 211, **290**
of a `record` class 311
overloaded **289**

constructor reference **559**
constructors cannot specify a return type 273
consume memory 928
consumer 699, **733**
consumer electronic device 7
`Consumer` functional interface 541, 549
consumer thread **733**
containerization **27**
`contains` method of `Collection` **462**
`contains` method of class `Array-List<E>` 200, **203**
`containsKey` method of `Map` **488**
content panel (Scene Builder) 571
context of a token in genAI **39**
context switching **699**
context window in generative AI **39**
contextual keywords 50
`continue` statement 50, 116, **117**
control statement 71, 72, 928

`for` 102
nesting **73**
stacking **73**
`switch` expression **141**

control variable **79**, 100, 101, 102
controller (in MVC architecture) 581
Controller (Scene Builder) 587, 588, 605
controller class (JavaFX) **571**, 581, 582, 587, 605

initialize instance variables 589

`initialize` method 593

controller skeleton 588
controlling expression of a `switch` **114**
Controls

`RadioButton` **601**
`TableView` **846**

controls 568, **570**
ControlsFX open source project 663
convenience factory method 456, **489** , 490
converge on a base case 920
conversion characters **1002**

`%` 1006
`A` 1004
`a` 1004
`B` **1006**
`b` 1006
`C` 1005
`c` 1005
`d` **1003**
`E` 1004, **1004**
`e` 1004, **1004**
`f` 1004
`G` **1004**
`g` **1004**, 1004
`H` **1006**
`h` 1006
`n` 1006
`o` **1003**
`S` **1005**
`s` **1005**
`X` **1003**
`x` **1003**

convert

a binary number to decimal 1023

a hexadecimal number to decimal 1023
an octal number to decimal 1023
between number systems 235

copy constructor 291
`copy` method of `Collections` 469, **475**
copying files 409
copying objects

deep copy **332**
shallow copy **332**

copyright (generative AI) 45
core package 53
corpus **239**, 249

corpora (plural) **239**

`cos` method of `Math` 153
cosine 153
`count` method of interface `IntStream` **536**
counter **79**
counter-controlled iteration **79**, 80, 85, 88, 100, 101, 928
counting loop 101
counting word frequencies **240**
country codes (ISO 3166-1) 167
`cp` option for specifying the classpath **422**
CraigsList 21
craps (casino game) 142
create a class that resides in a module 876
create a custom runtime 903
create an object of a class 210, 267
`create` method of `ChatCompletions` **800**
`create` method of `Images` **809**
`create` method of `Moderations` **812**
`create` method of `URI` **430**
create, read, update and delete (CRUD) **832**
`createStatement` method of `Connection` **844**
creating and initializing an array 173

creating files 409
crime prevention 31
CRISPR (Clustered Regularly Inter-spaced Short Palindromic Repeats) gene editing 31
crop yield improvement 31
CRUD operations (create, read, update and delete) **832**
cryptocurrency xlii, 24, 30, 95, 259
cryptography xli, xliv, xlvi, 24
    Advanced Encryption Standard (AES) **253**
    AES xli
    asymmetric-key **253**, 446
    ciphertext **253**
    key size in AES **257**
    plaintext 253
    post-quantum 7
    private-key **253**
    RSA xli
    secret key **253**, 253
    symmetric-key **253**, 253
CSS (Cascading Style Sheets) **569**, 604
    selector **642**
`.css` filename extension **641**
CSS property (JavaFX)
    `-fx-arc-height` **653**, 653, 687, 687
    `-fx-arc-width` **653**, 653, 687, 687
    `-fx-fill` **653**, 653
    `-fx-font` **643**, 643
    `-fx-font-family` **643**, 643
    `-fx-font-size` **643**, 643, 644, 644
    `-fx-font-style` **643**, 643
    `-fx-font-weight` **643**, 643
    `-fx-padding` **642**, 642
    `-fx-rotate` **682**, 682
    `-fx-spacing` **642**, 642

-fx-strikethrough **644**, 644
-fx-stroke **652**, 652
-fx-stroke-line-cap **653**, 653, 687, 687
-fx-stroke-width **652**, 652
-fx-underline **644**, 644

CSS property name (JavaFX) 642
CSS pseudo-class

:hover **682**

CSS rules **641**
CSS transition

ease timing function **683**
hovering the mouse over an element **680**
transition-delay **683**
transition-duration **683**
transition-property **683**
transition-timing-function **683**

CSS transitions (JavaFX) **680**
CSV (comma-separated values) xli

.csv file extension **432**
file format 406, **432**

CsvMapper class **434**, 436

readerFor method **436**
readValues method **436**
with method **436**

CsvSchema class **434**, 436

emptySchema method **439**
schemaFor method **434**
withHeader method **434**

Ctrl key 113

<*Ctrl*>-*d* 113

<Ctrl>-z 113

currency-formatted `String` 593
current folder (`.`) in a path 409
`current` method of class `ThreadLocalRandom` **703**
`currentThread` method of class `Thread` **709**
cursor 52, 54
cursor for a `ResultSet` **845**
custom character class **243**
custom JavaFX controls 632
custom precision for formatted floating-point values **86**
custom runtime 903, 904

create 903

customer

service agents 31

cycle in a module-dependency graph 893
cycle in module dependencies 893, 894
cycle indefinitely (animation) 675
`CycleMethod enum` **688**
`Cylinder` class **690**

D

`-d` option of the `javac` command 881
`-D` options to the Java runtime **166**
daemon thread 714
Dall-E xliv

AI image generator 760
API **786**

Dall-E image generator **32**
dApps (decentralized applications) 24
data analysis 30
data analytics xli, xlix, 407, 432, 437
data binding 620, **846**

data-carrier classes **310**
data integrity 297
data-interchange format
    JSON 418
data mining 209
data munging **209**, 248, **438**
data persistence **406**
data science xli, xlii, xlix, 31, 208, 407, 432, 438
    get to know your data 437, 440
    use cases 31
data source (stream) 523, 525, 529
data structure xlix, **170**, 456, 457
data visualization 31
data wrangling xli, **209**, 248, 438
database 830, 836
database management system (DBMS) **830**
database URL (connection string) 843
dataset 407, 432
    Titanic disaster 437
date formatting 1002
Date/Time API li, **155**, 162, 199, 286
`DateTimeFormatter` class **162**, 166
    `ofLocalizedDateTime method` **166**
`DayOfWeek enumgetDisplay-Name` method **165**
`DAYS` constant of the `ChronoUnit enum` **165**
Db2 (IBM) 830
DBMS (database management system) **830**
deadlock 755, 756
dealing 278
decentralized apps (dApps) **24**, 24
decentralized finance (DeFi) 24
decimal (base 10) number system **1018**
decimal integer 1003

decimal integer formatting 59
decision 62, 73
    symbol in the UML **73**
declaration
    class **50**
    `import` **57**
    method **51**
declarative programming **524**, 570
declare a variable in JShell 950
`decode` method of `Base64` **258**
decomposing `record` objects in patterns **314**
decrement a control variable **104**
decrement operator `--` **91**
decrypt
    ciphertext 258
    messages 252
`DECRYPT_MODE` constant of class `Cipher` **258**
deep copy **332**
deep learning xlix, **35**, 407, 432
DeepBlue **33**
`default` case in a `switch` **114**, 115
default constructor **273**, 296, 324
default exception handler 393
default initial value **267**
`default` interface methods (Java SE 8) 357
`default` keyword 50
default locale 591
`default` method in an interface **357**, 540
default package **270**
default upper bound (`Object`) of a type parameter 507
default value **267**
DeFi (decentralized finance) 24
definite iteration **79**
Deitel & Associates, Inc. lx

virtual and on-site corporate training lx

Deitel blog

https://deitel.com/blog xli

Deitel, Dr. Harvey M. lix
Deitel, Paul J. lix
delegating constructor **290**, 370
`DELETE FROM` SQL statement 836, **841**, 841
`delete` method of class `String-Builder` **231**
`deleteCharAt` method of class

`StringBuilder` **231**

deleting directories 409
deleting files 409
delimiter 238

for tokens **237**

dependency graph **880**, 893

cycles not allowed 893

dependency injection xli, xlviii, **359**, 361, 366, 875
dependency is not fulfilled 890
dependent condition **120**
`Deque` interface 466
descending sort

`DESC` 838

`--describe-module` option of the `java` command **877**, 882
descriptive statistics **441**, 441, 442
deserialization **350**, 406, 419

CSV 436

design area 571
design pattern **26**
design process **11**
deterministic random numbers **142**

`DeveloperMessage` class **800**
device driver 337
diagnostic medicine 31, 32
`DialogPane` class (JavaFX) 599
diamond in the UML **71**
    decision symbol **73**
    merge symbol **78**
diamond notation in generics(`<>`) **203**, 463
dice game 142
difference between values (random numbers) 142
digit 58, 236, 1018
`digit` method of class `Character` **235**
digit separator (_) **58**, 139
digital ownership 24
direct subclass **321**
direct superclass 321
directed edge in a module-dependency graph **880**
`Directional` class 691
directories 409
    creating 409
    getting information 409
    manipulating 409
`DirectoryChooser` class
    `showDialog` method **630**
`DirectoryChooser` class (JavaFX) 598, **625**, 629
`DirectoryStream` interface **409**
discover module dependencies 896
`disjoint` method of `Collections` 469, **479**
display a line of text 52
`distinct` method of interface `Stream` **554**
divide-and-conquer approach 920
divide by zero 15, 379
division 60, 61

compound assignment operator, /= 91

do...while iteration statement 50, 72, **109**
document a program 49
doFinal method of class Cipher **258**, 259
dollar sign ($) 51, 58
dot (.) separator 153
DOT (.dot) files 897
DOT graph description language 897
dot operator (.) **67**, 97, 108, 152, 187, 304
dotted line in the UML **72**
(double) cast **85**
Double class **458**, 513

isNaN method **441**

double keyword 134
double-precision floating-point number **83**
double primitive type 50, **58**, 83

promotions 160

double quote, " 51, 52, 55
double selection statement **72**
DoubleProperty class (JavaFX) 593, **616**

asString method **616**

doubles method of class

SecureRandom **561**

DoubleStream interface **534**, 556

average method 557
reduce method 557
sum method 557

doubleValue method of Number **514**
downcasting **346**
downstream Collector **556**
drag the mouse to draw 600

driver class **266**
driver program **65**
`DriverManager` class **844**
`getConnection` method **844**
`/drop` JShell command 978
`dumpStack` method of class `Thread` 379
`Duration` class
`between` method **164**, **702**
`java.time` package **162**
`toHours` method **165**
`toMillis` method **702**
`toMinutes` method **165**
`toSeconds` method **165**
dynamic
driving routes 31
dynamic binding **336**, 345
dynamically resizable **171**

E

eager **527**
`ease` timing function **683**
`EASE_BOTH` interpolator **673**
`EASE_IN` interpolator **674**
`EASE_OUT` interpolator **674**
Eclipse IDE for Java Developers lxiv, lxv, 13
edge computing **21**
`/edit` JShell command **956**, 978
editor 12
EEPs (examples, exercises and projects) xliii
effectively `final` local variables **400**, 542
electronic health records 31
element (XML) **646**

element of an array **171**
element of chance **137**
eligible for garbage collection 306
`Ellipse` class (JavaFX) **649**, 651

- `centerX` property **651**
- `centerY` property **651**
- `radiusX` property **651**
- `radiusY` property **651**

ellipsis (. . .) in a method parameter list **193**
`else` keyword 50, 74
e-mail (electronic mail) 19
embedded system 5
emotion detection 31
`Employee` abstract base class 338
`Employee` abstract superclass 339
`Employee` class hierarchy test program 344
`Employee` class that implements `Payable` 353
empty element 647
empty statement (a semicolon, `;`) 64, 76, 103
empty string 67, 211
`emptySchema` method of `CsvSchema` **439**
enable assertions 398
`--enable-preview` option (`javac` and `java` commands) **724**
encapsulation **10**
enclosing folder (. .) in a path 409
`encode` method of `Base64Util` **803**
`encode` method of `URLEncoder` **429**
`encodeToString` method of `Base64` **255**
`encoding` attribute in `xml` declaration **646**
encrypt plaintext 258
`ENCRYPT_MODE` constant of class `Cipher` **258**
encrypting messages 252
“end of data entry” 83
end-of-file (EOF)

- indicator **113**

marker 407

end-of-line (single-line) comment, // 49, 52
end tag 646
endsWith method of class String 68, 216
endX property of a Line 651
endY property of a Line 651
enhanced for statement 184
ensureCapacity method of class

StringBuilder 226

Enter (or *Return*) key 52
enter an expression in JShell 953
entering text in a text field 580
entity-relationship (ER) diagram 835, 835
entry method of interface Map 491
entry point of a control statement 73
entry point of an application 882, 883, 884, 889, 908
enum 145

constant 300
constructor 301
declaration 300
EnumSet class 303
keyword 50, 145
values method 302

enum type 145
enumeration constant 145
EnumSet class 303

range method 303

/env -add-modules JShell command 975
/env -class-path JShell command 975
/env -module-path JShell command 975
/env JShell command 978
environment variablePATH_TO_FX lxvi, 578
equal chance (or probability) 138

equal likelihood 139
equality operators **62**

    `==` to compare `String` objects 213

`equals` method

    of a `record` class 312
    of class `Arrays` **197**, 199
    of class `Object` 332
    of class `String` **67**, 213, 214

`equalsIgnoreCase` method of class `String` **214**
erasure **501**, 504, 507
`Error` class **385**
escape character **55**, 840
escape sequence **55**, 58, 1013

    `\` backslash 55
    `\"` double-quote 55
    `\n` newline 55
    `\t` horizontal tab 55

escaping a backslash **242**
Ether (cryptocurrency of Ethereum platform) 25
Ethereum 24
event 357, **571**, 580
event driven **580**
event handler 357, **571**, 580
event handling 571, 580, **580**
event listener 357

    interface **581**

event queue 741
`EventHandler<ActionEvent>` interface (JavaFX) **592**

    `handle` method **592**

exabytes (EB) 29
exaflops 6

examples, exercises and projects (EEPs) xliii
exception **183**, 376
    `ArrayIndexOutOfBoundsException` 183, 184
    checked 397
    handler **183**
    handling 183
    parameter 184
    supressed **400**
    translation 396
    unchecked 397
`Exception` class **384**
exception handler **382**
exception parameter **382**
execute **14**, 52
execute a multiple-source-file program with the `java` command 18
execute a single-source-file program with the `java` command 18
execute a statement in JShell 949
`execute` method of the `Executor` interface **707**, 711
execute the last valid snippet (JShell) 953
`executeQuery` method
    of `PreparedStatement` **856**
    of `Statement` **844**
`executeUpdate` method of interface `PreparedStatement` **857**
execution-time error **15**
`Executor` interface **707**
    `execute` method **707**, 711
`Executors` class **707**
    `newCachedThreadPool` method **709**
    `newVirtualThreadPerTaskExecutor` method **713**
`ExecutorService` class 713

`awaitTermination` method 715
`close` method 715
`shutdown` method 715

`ExecutorService` interface **707**, 717

`awaitTermination` method **710**, 715
`close` method **710**
`shutdown` method **710**
`submit` method **717**

Exercises

Files Generative AI: Apache Commons CSV Library 454
Files Generative AI: Best Practices for Processing Large Files 454
Files Generative AI: Binary File Processing 454
Files Generative AI: Cryptography Algorithm Comparison 454
Files Generative AI: Elliptic Curve Cryptography 454
Files Generative AI: File Comparison 454
Files Generative AI: `Files` Class Tutorial 452
Files Generative AI: OpenCSV Library 454
Files Generative AI: `PrintWriter` vs. `Formatter` for Formatted Output 453
Files Generative AI: Storing RSA Private Keys Securely 454
Files Generative AI: Tablesaw Library 453
Files Generative AI: Telephone-Number Word Generator 453
Generative AI

JavaFX `MediaPlayer` Errors 696

Generative AI: Advantages and Disadvantages of Custom Exceptions 402
Generative AI: `ArrayList` vs. `LinkedList` 493
Generative AI: Challenges of Processing Monetary Amounts 129
Generative AI: Checking Java Idiom 205

Generative AI: Choosing the Right Initial Size of a Collection 493
Generative AI: `Collections` Methods for Creating Singletons 493
Generative AI: Collections Optimized for Primitive Types 493
Generative AI: Constructor Injection vs. Property Injection 373
Generative AI: Control Statements Exercises 128
Generative AI: `default` Methods 373
Generative AI: Do Organizations Discourage Exceptions 402
Generative AI: Exception-Handling Performance 403
Generative AI: Exceptions and Generics 518
Generative AI: Exceptions Thrown By `BigDecimal` Methods 403
Generative AI: Failing Fast vs. Exception Handling 205
Generative AI: `finally` Blocks 402
Generative AI: Format String Safety 1015
Generative AI: Formatting Dates and Times 1015
Generative AI: Generative AI: Composition vs. Inheritance 373
Generative AI: Generic `maximumElement` and `minimumElement` Methods 518
Generative AI: `HashSet` vs. `SortedSet` 493
Generative AI: Identifying a JavaFX Application's Model, View and Controller 596
Generative AI: Improving Fault Tolerance 205
Generative AI: Java Collections Performance Comparison 493
Generative AI: JavaFX 3D Lighting and Camera Effects 696
Generative AI: JavaFX Application Lifecycle Events 637
Generative AI: JavaFX `AudioClip`s 696
Generative AI: JavaFX Bidirectional Property Bindings 636
Generative AI: JavaFX Demos 595
Generative AI: JavaFX Effects 596

Generative AI: JavaFX FXML Scrap-book Layout 595
Generative AI: JavaFX Image Effects 696
Generative AI: JavaFX Internationalization and Localization 637
Generative AI: JavaFX `MediaPlayer` Events 696
Generative AI: JavaFX `Node` Transformations 596
Generative AI: JavaFX Node Transformations 696
Generative AI: JavaFX Painter Application Enhancements 636
Generative AI: JavaFX Property Bindings vs. Property Listeners 636
Generative AI: JavaFX Responsive Design 637
Generative AI: JavaFX Tip Calculator User Interface Refactoring 595
Generative AI: JavaFX Touch Events and Gestures 637
Generative AI: Locale-Specific Formatting 1015
Generative AI: Logging Exceptions 403
Generative AI: Making a Difference 205
Generative AI: Method `maximum` with a Variable-Length Argument List 518
Generative AI: Modernizing Java Code 206
Generative AI: One-Dimensional Array Animation Demos 205
Generative AI: One-Dimensional Array Practice Exercises 205
Generative AI: Overloaded Generic Method `printArray` 518
Generative AI: Passing Arguments to Methods 206
Generative AI: Postconditions 402
Generative AI: Primitive vs. Reference Types 206
Generative AI: Processing Monetary Amounts Precisely in Java 129
Generative AI: Research the Next Version of Java 205
Generative AI: Runtime Polymorphism vs. Compile-Time Polymorphism 373
Generative AI: Swallowing Exceptions 403
Generative AI: Two-Dimensional Array Animation Demos 205
Generative AI: Two-Dimensional Array Practice

Exercises 205
Generative AI: Using `RuntimeException`s to Prevent "`throws` Chains" 403
Generative AI: What Does This One-Dimensional Array Code Do? 205
Generative AI: What Does This Two-Dimensional Array Code Do? 205
JPMS: Best Practices 916
JPMS: Granting Temporary Access to a Module 917
JPMS: `java.base` Module 917
JPMS: `jpackage` Tool 917
JPMS: JPMS and Java Security 917
JPMS: Modular Software Design 917
JPMS: Multi-release JARs 917
JPMS: Pitfalls 916
JPMS: Real-World Use-Cases of `ServiceLoader` 917
JPMS: Security Best Practices 916
JPMS: Step-By-Step Modularization Migration Guide 917
JPMS: Troubleshooting Tips 916
JPMS: Benefits and Trade-Offs 916
Lambdas and Streams GenAI:`Collectors` 566
Lambdas and Streams GenAI:Exception Handling with Streams 565
Lambdas and Streams GenAI:Immutability 565
Lambdas and Streams GenAI:Loop vs. Stream Performance 565
Lambdas and Streams GenAI:`Optional` Objects 565
Lambdas and Streams GenAI:Refactoring Collection Code 565
Lambdas and Streams GenAI:`Stream` Method `mapMulti` 566
Lambdas and Streams GenAI:Stream Processing of Stack Traces with `StackWalker` 565
Lambdas and Streams GenAI:Streams Best Practices 565

Lambdas and Streams GenAI:Streams Pitfalls 566
Lambdas and Streams GenAI:`switch` Expressions in Stream Pipelines 566
Lambdas and Streams GenAI:Testing and Debugging Stream Pipelines 566
Project Loom: Generative AI Exercises 755, 1015

`exists` method of class `Files` **411**
`/exit` JShell command **959**, 978
`exit` method of class `System` **388**
exit point
of a control statement 73

exiting a `for` statement 117
`exp` method of `Math` 153
explicit conversion **85**
explicit dependencies in a module graph 880
exploded-module folder **881**
exponential format 1002
exponential method 153
exponential notation **1004**
exponentiation operator 108
`exports` keyword 50
`exports` module directive 873, **874**, 886, 890
`exports...to` module directive **874**
expression **59**
expression statement **952**
extend a class **320**
extended reality (XR) **23**
`extends` keyword 50, **322**, 326
extensible 333
language **263**

external iteration **522**, 561
external snippet editor (JShell) 982
extracting data from text 241
extracting key points 768

F

facial recognition 31
factorial 921
`factorial` method 921, 922
factory method **370**, 456, 489
`factory` method of class

`Thread.Builder.OfVirtual` **716**

`FadeTransition` class **673**
fail fast iterator **462**
failed state of a `Task` 746
fair rounding algorithm 124
`false` keyword **62**, 73
fatal

runtime error **15**

fatal error **76**

fatal logic error **76**

fault tolerant 59, **183**, 376
features in a dataset **437**
feedback mode (JShell) **955**, 980
`fibonacci` function 925
Fibonacci series 742, 924, 926

defined recursively 924

field

default initial value **267**
of a class 148, **154**
width **107**

field width 1002, 1003, 1007
`File` class

`toPath` method **629**
used to obtain file and directory information 410

`file` method of `TranscriptionRequestBuilder` **806**
`FileChooser` class 598, **625**, 627

`showOpenDialog` method **629**
`showSaveDialog` method 629

`FileNotFoundException` class **415**
`FileOutputStream` class

`write` method **808**

files 406

copying 409
creating 409
getting information about 409
manipulating 409
reading 409

`Files` class **409**, 558

`exists` method **411**
`getLastModifiedTime` method **412**
`isDirectory` method **412**
`lines` method **558**
`newDirectoryStream` method **412**
`readString` method **766**
`size` method **412**

fill color 614
`fill` method

of class `Arrays` **197**, 199
of class `Collections` 469, **475**

`fillArc` method of class `Graphics-Context` **689**
`fillOval` method of class `Graphics-Context` **688**
`fillRect` method of class `Graphics-Context` **687**
`fillRoundRect` method of class `GraphicsContext` **687**
`FillTransition` class **672**
filter a stream 529

`filter` method of interface `IntStream` **529**
`filter` method of interface `Stream` 544, 545, 547
filtering elements of a collection 462
`final`
    class **348**
    keyword 50, 116, **154**, 176, 308, 347, 732
    method **347**
    variable 176
final state in the UML **72**
final value 100
`finally`
    block 382, **388**
    clause 50, **388**
    keyword 382, **388**
`find` method of class `Matcher` **246**, 248
`FindException` class 904, 905
`findFirst` method of interface `Stream` 551
Firefly image generator 32
Firefly video generator 32
`first` method of `SortedSet` **486**
`firstContent` method of `Chat` **800**
first-in-first-out (FIFO) 741
Fisher-Yates shuffling algorithm 281
`Fit Width` property of an `ImageView` (JavaFX) 575
fixed text 59
    in a format string **56**, 1002
fixed thread pool 718
fixed-point format (floating-point numbers) 1004
flag value **83**
flags 1009
`flatMap` method of interface `Stream` **559**
`float`
    primitive type 50, **58**, 83
    primitive type promotions 160

`Float` class **458**
floating-point conversion specifiers 1008
floating-point number **81**, 83, 85, 1005

- custom precision for formatting **86**
- literal **83**
- literals are `double` by default 83
- division 86
- double precision **83**
- single precision **83**
- `%f` format specifier **86**
- fixed-point format 1004

`floor` method of `Math` 153
FLOPS (floating-point operations per second) 6
flow of control **71**, 85

- `if` statement 73
- `if`...`else` statement 74
- `while` statement 78

`FlowPane` class (JavaFX) 599
focus **586**
folder in a path

- current (`.`) 409
- enclosing (`..`) 409

`Font` property of a `Label` (JavaFX) 574
`for` iteration statement 50, 72, **101**, 102, 108

- enhanced **184**
- header **103**
- nested 179, 192

`forDigit` method of class `Character` **235**
`forEach` method of interface `IntStream` **533**
`forEach` method of interface `Map` **555**
foreign key **834**, 836
`fork` method of class `StructuredTaskScope` **727**
Fork/Join framework 755

`format` method

of class `Formatter` **1014**
of class `NumberFormat` 703
of class `String` **284**, 1014, **1014**

`format` method of class `LocalDate-Time` **166**
`format` method of class `NumberFormat` **593**
format specifiers **56**, 1002

`%.2f` for floating-point precision 86
`%%` 1006
`%B` 1006
`%b` **67**, 122, 1006
`%c` 1006
`%d` **59**, 1002, 1003
`%E` 1005
`%e` 1005
`%f` **86**, 151, 1005
`%G` 1005
`%g` 1005
`%H` 1006
`%h` 1006
`%n` 1006
`%n` (line separator) **56**
`%o` 1004
`%S` 1006
`%s` **56**, 1002, 1006
`%X` 1004
`%x` 1004

format string **56**, 1002, 1008
`FormatStyle enum` (`java.time.format` package) **162**

`MEDIUM` **166**

formatted

text block 1015

formatted output 1006

, (comma) formatting flag **108**
%f format specifier 151
- (minus sign) formatting flag **107**
0 flag **179**, 284
aligning decimal points in output 1002
boolean values 122
comma (,) formatting flag **108**
conversion character 1002
dates 1002
display formatted data 56
exponential format 1002
field width **107**, 1002
floating-point numbers 151
Formatter 1014
grouping separator **108**
inserting literal characters 1002
integers in hexadecimal format 1002
integers in octal format 1002
left align **107**, 1002
locale-specific **161**, 161
minus sign (-) formatting flag **107**
precision 1002
right align **107**, 1002
rounding 1002
times 1002

formatted String 1014, 1015
Formatter class **408**, 413, 1002, 1014

close method **415**
format method **415**, 1014
toString method 1014

FormatterClosedException class **415**
forward referencing an undeclared method (JShell) **972**
forward slash character (/) in end tags **647**
four V's of big data 30
fractal **931**

depth **932**
Koch Curve **931**
Koch Snowflake **932**
level **932**
order **932**
self-similar property **931**
strictly self-similar fractal **931**, 936

fragile software **371**
frame rate 678
fraud detection 31
`frequency` method of `Collections` 469, **479**
`FROM` SQL clause 836
`FULL` constant of the `TextStyle enum` 165
fully qualified

class name 270

`Function` functional interface 541, 547

`identity` method 562

functional interface **358**, 540, 581, 594

`BiConsumer` **556**, 562
`BinaryOperator` 540
`Consumer` 541, 549
`Function` 541, 547
`@FunctionalInterface` annotation **564**
`IntBinaryOperator` 540
`IntFunction` 541
`IntFunction<T>` 701
`IntPredicate` 541
`IntToDoubleFunction` 701
`IntToLongFunction` 701
`IntUnaryOperator` 541, 701
`Predicate` 541, 550
`Supplier` 541
`UnaryOperator` 541

functional programming xlv, 520, **524**, 703

pure function **540**

functional-style programming 4
`@FunctionalInterface` annotation **564**
`Future` interface **717**

`get` method **717**

fx:id property of a JavaFX component **582**
`FXCollections` class (JavaFX) **617**

`observableArrayList` method **620**

FXGL (FX Game Library) xlii
FXGL (game library) **694**

animations and sprites 694
collision detection 694
game state management 694
multiplayer capabilities 694
particle effects 694
physics engine 694
player input handling 694
rendering 694
scene transitions 694
sound effects 694

FXML (FX Markup Language) **569**

`import` declaration **646**
local resolution operator 648

`@FXML` annotation 591, **591**
`@FXML` instance variables 607
`FXMLLoader` class (JavaFX) **581**, 589, 592, 607

`load` method **581**, 589

G

game
development 694
playing 137
state management 694
game development
card games 278
craps (casino game) 142
guess-the-number game 15
multiplayer capabilities 694
simulation 137
game playing 137
garbage collector **303**, 384, 388, 699
Gary Kasparov 33
Gemini 36
Gen-3 Alpha video generator 32
genAI
prompt engineering guide 40
sample prompts 40
general class average problem 83
generalities 333
`generate` method of interface `IntStream` **564**
`generateKey` method of class `KeyGenerator` **257**
`generateKeyPair` method of `KeyPairGenerator` **447**
generative AI xxvii, xxxiii, **35**, 136, 237, 240, 408, 413
Anthropic Claude lxvii
API-based applications lii, 758
bias 45
ChatGPT lxvii
Claude lxvii
context window **39**
copyright 45
Gemini lxvii
genAI lxvii

Google Gemini lxvii
hallucination **40**
multimodal **758**, 759
OpenAI ChatGPT lxvii
Perplexity lxvii
prompt engineering xxxiv, **39**, 39
self-attention **38**
token **39**
transformer **38**

Generative Pre-Trained Transformer (GPT) large language model 38
generator **701**
generic `ArrayList` collection xlix
generic data structures xlix
generics 457, **496**

`?` (wildcard type argument) **515**
actual type arguments **499**
angle brackets (`<` and `>`) **499**
class **200**, 496, 505
default upper bound (`Object`) of a type parameter 507
diamond notation 463
erasure **501**
interface 457, **502**
method **496**, 498, 505, 804
parameterized type **505**
scope of a type parameter 507
type erasure **501**
type parameter **499**
type parameter section **499**
type variable **499**
upper bound of a type parameter **503**, 504
upper bound of a wildcard 515
wildcard type argument **515**
wildcard without an upper bound 516
wildcards **513**, 515

`get` method

of a `record` class 312
of class `ArrayList<E>` **203**
of interface `Future` **717**
of interface `List<T>` **461**
of interface `Map` **488**
`StructuredTaskScope.Subtask` interface **728**

get method 297
`get` method of class `ScopedValue` **727**
`GET` method of `HttpRequest.Builder` **430**
get to know your data 437, 440
`getAsDouble` method of class `OptionalDouble` **536**, 557
`getAsInt` method of class `Option-alInt` **536**
`getAutoCommit` method of interface `Connection` **865**
`getBytes` method of class `String` 257
`getCause` method of `Throwable` **395**
`getChars` method
of class `String` **211**
of class `StringBuilder` **227**
`getChildren` method of a layout pane (JavaFX) **608**
`getClass` method of `Object` 332, **346**
`getClassName` method of class `StackTraceElement` **393**
`getColumnCount` method of `ResultSetMetaData` **845**
`getColumnType` method of `ResultSetMetaData` **845**
`getConnection` method of `Driver-Manager` **844**
`getCurrencyInstance` method of class `NumberFormat` **591**
`getDayOfWeek` method of class `LocalDate` **164**
`getDays` method of `Period` class **165**
`getDecoder` method of `Base64` **258**
`getDefault` method of class `Locale` **165**
`getDefault` method of `RandomGenerator` **137**
`getDisplayName` method of `DayOf-Week enum` **165**
`getEncoded` method of `SecretKey` **255**
`getEncoder` method of `Base64` **255**
`getenv` method of `System` **429**
`getFileName` method of class `Stack-TraceElement` **393**

getFileName method of interface Path 411
getFirst method
    of ArrayList<E> 200
getFirst method of ArrayList 442
getGraphicsContext2D method of class Canvas 686
getInstance method
    of KeyPairGenerator 447
getInstance method of class Cipher 258, 258
getInstance method of class Key-Generator 257
getInt method of interface ResultSet 845
getLast method
    of ArrayList<E> 200
getLast method of ArrayList 442
getLastModifiedTime method of class Files 412
getLineNumber method of class StackTraceElement 393
getMessage method of class Throwable 393
getMethodName method of class StackTraceElement 393
getMonths method of Period class 165
getName method of class Class 332, 346
getObject method of interface ResultSet 845
getPercentInstance method of class NumberFormat 591, 703
getProperty method of System 412
getPublic method
    of KeyPair 447
getSegments method of Transcription 811
getSelectedToggle method of class ToggleGroup 608
getSelectionModel method of class ListView 621
getSimpleName method of Class<T> 805
getStackTrace method of class Throwable 393
getSurpressed method of an exception 400
getText method of class TextInput-Control 592, 592, 593
getting questions answered lv
getUserData method of interface Toggle 609

`getValue` method of class `Color` class

`getValue` method **940**

`getValue` method of interface `Worker` **746**
`getWidth` method of class `Canvas` class

`getWidth` method **940**

`getYears` method of `Period` class **165**
gigabytes (GB) 29
gigaflops 6
Git version control system **26**
GitHub xxxiii, liv, lvi, 11, 26
Go board game 34
golden mean (Fibonacci) **924**
golden ratio (Fibonacci) **924**
Google 759
Google Assistant 239
Google Cloud Platform (GCP) 20
Google Gemini lxvii, 36, 759, 813
Google Imagen image generator 32
Google Maps 21
Google Search lviii
Google Veo 2 video generator 821
Google Veo video generator 32
Gosling, James 7
`goto` statement **70**
GPT (Generative Pre-Trained Transformer) large language model 38
gradient **653**

linear **654**

Gradle build tool 888
grammar

checking **240**
correction **240**

Grammarly lviii

graph description language DOT 897
graph information 179
graphical user interface (GUI) 357, 568
graphics xlii
`GraphicsContext` class **685**

- `fillArc` method **689**
- `fillOval` method **688**
- `fillRect` method **687**
- `fillRoundRect` method **687**
- `setFill` method **687**
- `setGlobalAlpha` method **687**
- `setLineWidth` method **686**
- `setStroke` method **686**
- `strokeArc` method **689**
- `strokeLine` method **686**, 687
- `strokeOval` method **688**
- `strokeRect` method **687**
- `strokeRoundRect` method **687**

Graphviz 897
greedy quantifier **244**
`GridPane` class (JavaFX) **579**, 579, 580, 599

- add rows or columns 583
- Hgap property 586
- Pref Height property 603
- Pref Width property 603
- Row Span property of a control 613
- Vgap property 586

`GROUP BY` SQL clause 836
group digits for readability 58
`group` method of class `Matcher` **247**, 248
grouping of operators **61**
grouping separator (formatted output) **108**
`groupingBy` method of class `Collectors` **555**, 556, 559, 562
GStreamer framework 633
guard condition in the UML 73

GUI (Graphical User Interface) xlii, 357
GUI (Graphical User Interface) component **568**

- `BorderPane` (JavaFX) **601**
- `BorderPane` class (JavaFX) **601**
- `Button` (JavaFX) **570**, 580
- `ImageView` (JavaFX) **571**
- `Label` (JavaFX) **570**, 571
- naming convention 582
- `Slider` (JavaFX) **570**, 577, 580, **580**
- `TextField` (JavaFX) **570**, 580

guillemets (« and ») **274**

H

`H` conversion character **1006**
`h` conversion character **1006**
half-even rounding **127**
half-up rounding **127**
half-open range of integers 523
hallucination (genAI) xxxvi, **40**
handle an exception 380
`handle` method of class `AnimationTimer` **678**
`handle` method of interface `EventHandler` **592**
has-a relationship **297**, 319
hash bucket **482**
hash table 482, 483

- collisions 482, 483

`hashCode` method of class `Object` 332, 483
hashing **482**
`HashMap` class **486**

- `keySet` method **488**

`HashSet` class **483**
`hasNext` method

of class `Scanner` **114**
of interface `Iterator` **461**, 465

`hasPrevious` method of `ListIterator` **466**
`HBox` class (JavaFX) 599
`headSet` method of class `TreeSet` **485**
health outcome improvement 31
`height` property of a `Rectangle` **651**
helper method **286**

in an interface 358

hexadecimal (base 16) number system **1018**
hexadecimal integer 1003
hexadecimal literal value 1023
Hgap property of a `GridPane` 586
hidden fields 148
hide implementation details 286
hierarchy

of employee types 322

Hierarchy (Scene Builder) 575, 603
High DPI (dots-per-inch) screens 633
`/history` JShell command **951**, 978
horizontal tab 55
hot spots in bytecode 15
HotSpot compiler **15**
`:hover` CSS pseudo-class **682**
hovering the mouse over an element **680**
HTML (HyperText Markup Language) **19**
HTTP (HyperText Transfer Protocol) **19**
HTTP request method GET 430
`HttpClient` class 406, 430

`newHttpClient` method **430**

`HttpRequest` class 406
`HttpRequest.Builder` class **430**

`build` method **430**

`GET` method **430**
`uri` method **430**

`HttpResponse` class 406

status code 430

`HttpResponse.BodyHandlers` class

`ofString` method **430**

HTTPS protocol 94, 252
Hugging Face 759, 813
human genome sequencing 31
HyperText Markup Language (HTML) **19**
HyperText Transfer Protocol (HTTP) **19**

I

IBM Db2 830
IBM DeepBlue **33**
IBM Watson **33**
ID selector (JavaFX) **643**, 644
IDE (integrated development environment) **13**
identifier **51**, 58

camel case naming 134, 264

identity column 853
`identity` method of functional interface `Function` 562
identity theft prevention 31
identity value in a reduction **537**
IDEs

Eclipse IDE for Java Developers lxiv, lxv, 13
IntelliJ IDEA Community Edition lxiv, 13
Visual Studio Code lxiv, lxv, 13

`if` single-selection statement 50, **62**, 72, 73
`if`...`else` double-selection statement 72, **74**, 74
`ifPresent` method of class `Optional` **551**

ignoring array element zero 183
`IllegalArgumentException` class **284**
`IllegalStateException` class **417**
`Image` class (JavaFX) **621**
`Image` property of a `ImageView` 575
Imagen image generator 32
`image-pattern` CSS function **654**, 688
`ImageRequest` class **809**
    `builder` method **809**
`ImageRequestBuilder` class **809**
    `model` method **809**
    `n` method **809**
    `prompt` method **809**
    `quality` method **809**
`Images` class `create` method **809**
`images` method of `SimpleOpenAI` **809**
`ImageView` class (JavaFX) **571**
    `Fit Width` property 575
    `Image` property 575
    `setImage` method **621**
imitation game **36**
immutability **521**
immutable **210**, 221
    collections 489
    data 723, **732**
    object 414
imperative programming **524**
implement an interface **10**, 319, 350, 354
implementation-dependent code 286
implementation inheritance 359
`implements` keyword 50, **350**
implicit conversion **85**
implicit dependencies in a module graph 880

implicit variable in JShell **953**
implied readability **873**
`import` declaration 50, **57**, 270
import types into JShell 974
`/imports` JShell command **974**, 978
in parallel **698**
increment 103

- a control variable **100**, 101
- expression 117
- of a `for` statement 104

indefinite postponement 756
indentation 74, 75
index (subscript) **172**, 183
index `0` **172**
index validation (bounds checking) 181
`indexOf` method of class `Array-List<E>` 200, **203**
`indexOf` method of class `String` **218**
`IndexOutOfBoundsException` class 475
indirect recursive call **923**
indirect superclass 321
infer

- a local variable's type 135, 954
- a type with the diamond (`<>`) notation **203**
- parameter types in a lambda 527, 541
- return type of a lambda 526

infinite loop 103, 105, 923
infinite recursion **330**, 923, 928
infinite stream 562, **563**
infinity symbol 835
information hiding **10**, 270
inheritance **10**, 318

- `extends` keyword **322**, 326
- hierarchy **320**
- implementation 359

multiple 321
single **321**

`init` method of class `KeyGenerator` **257**
initial module **905**
initial state in the UML **72**
initial value of control variable **100**
initialize a controller's instance variables 589
initialize a variable in a declaration 58
`initialize` method

of `KeyPairGenerator` **447**

`initialize` method of a controller 608
`initialize` method of a JavaFX controller class 593
initializing two-dimensional arrays in declarations 191
inlining method calls **294**
inner class, anonymous **679**
`INNER JOIN` SQL clause 836, **839**, 839
innermost set of brackets 182
`input` method of `ModerationRequestBuilder` **811**
`input` method of `SpeechRequest-Builder` **807**
input/output operation 71
input/output package 155
`InputMismatchException` class **379**, 382, 417
`InputStream` class

`readAllBytes` method **808**

`INSERT INTO` SQL statement 836, **839**, 839
`insert` method of class `String-Builder` **230**
inserting literal characters in the output 1002
Inspector (Scene Builder) 573
instance **9**
instance `main` 51
instance method **136**, 265
instance variable **10**, 264, 274
`instanceof` operator 50

pattern matching **314**, 346, 365

`Instant` class **702**
instant feedback (JShell) liii
`instant` method of interface `InstantSource` **702**
`InstantSource` interface **701**

- `instant` method **702**
- `system` method **701**

`int` data type 91
`int` primitive type 50, **58**, 111

- promotions 160

`IntBinaryOperator` functional interface 540, 541
integer **57**

- array 175
- division **61**, 81
- quotient 61
- super-sized 94
- value 58

`Integer` class 196, **458**, 513

- `parseInt` method 196, 458

integer conversion characters **1003**
integral expression 116
integrated development environment (IDE) **13**
intelligent assistants xliv, 31, **239**

- Amazon Alexa xliv, 239
- Anthropic Claude 239
- Apple Siri xliv, 239
- Google Assistant 239
- Microsoft Copilot 239
- OpenAI ChatGPT 239

intelligent consumer electronic device 7
IntelliJ IDEA Community Edition lxiv, 13
interest rate 106

interface **10**, 319, 349, 351, 356

- `AutoCloseable` 357, **400**, 843
- `BiConsumer` **556**, 562
- `BinaryOperator` 540
- `BlockingQueue` **734**
- `Callable` **717**
- `CallableStatement` **864**
- `ChangeListener` (JavaFX) **581**, 593
- `Collection` **457**, 457, 468
- `Collector` **544**
- `Comparable` 356, **469**, 502
- `Comparator` **469**, 471
- `CompletableFuture` 797
- `Connection` **844**, 845
- `Consumer` 541, 549
- declaration **351**
- `default` methods (Java SE 8) **357**
- `Deque` 466
- `DirectoryStream` **409**
- `DoubleStream` **534**
- `EventHandler<ActionEvent>` (JavaFX) **592**
- `Executor` **707**
- `ExecutorService` **707**, 717
- `Function` 541, 547
- functional **358**
- `Future` **717**
- `InstantSource` **701**
- `IntBinaryOperator` 540
- `IntFunction` 541, 701
- `IntPredicate` 541
- `IntStream` **522**, 534
- `IntToDoubleFunction` 701
- `IntToLongFunction` 701
- `IntUnaryOperator` 701
- `IntUnnaryOperator` 541
- `InvalidationListener` 637
- `Iterable` 356

Iterator **457**
List **414**, **457**, 459, 466
ListIterator **459**
LongStream **534**
Map **457**, 486
Map.Entry 560
marker **350**
MultipleSelectionModel (JavaFX) **621**
ObservableList (JavaFX) **617**
ObservableValue (JavaFX) **593**
Path **409**
Predicate 541, 550
PreparedStatement 864
private methods (Java SE 9) **358**
Queue **457**, 480
RandomGenerator **137**
ResultSet **844**
ResultSetMetaData **845**
Runnable 357, **667**, 707
SAM (single abstract method) interface 358
sealed **367**
SecretKey 451
Set **457**, 483
SortedMap **486**
SortedSet **484**
Statement 845
static method 358, **358**
Stream 542
StructuredTaskScope.Subtask **728**
Supplier 541
tagging **350**
to a hierarchy 338
ToDoubleFunction **557**
Toggle **609**
UnaryOperator 541
Worker **741**

interface keyword 50, **351**

interlanguage translation **239**
intermediate stream operation 526, 529

- `filter` method of interface `IntStream` **529**
- `filter` method of interface `Stream` 544, 545
- `flatMap` method of interface `Stream` **559**
- `map` method of interface `IntStream` **526**
- `map` method of interface `Stream` 547
- `mapToObj` method of interface `IntStream` **533**, 535
- `sorted` method of interface `IntStream` **539**
- `sorted` method of interface `Stream` **544**, 545, 547

internal APIs **894**
internal iteration **524**
internationalization 580
Internet **18**, 424

- bandwidth 19

Internet domain name in reverse order (naming convention) 877
Internet of Things (IoT) 7, 20, 31
Internet Protocol (IP) **19**
`Interpolator` class (JavaFX) **673**

- `EASE_BOTH` interpolator **673**
- `EASE_IN` interpolator **674**
- `EASE_OUT` interpolator **674**
- `LINEAR` interpolator **673**

`interrupt` method of class `Thread` **708**
`InterruptedException` class **708**
`IntFunction` functional interface 541
`IntFunction<T>` functional interface 701
`IntPredicate` functional interface 541

- `and` method **553**
- `negate` method **553**
- `or` method **553**

`ints` method

RandomGenerator **532**, 561
SecureRandom **532**

ints method of class SecureRandom 561
IntStream interface **522**, 534

average method **536**
boxed method **562**
collect method **534**, 534, 535
count method **536**
filter method **529**
forEach method **533**
generate method **564**
iterate method **563**
limit method **563**
map method **526**
mapToObj method **533**, 534, 535
max method **536**
min method **536**
of method **535**
range method **523**
rangeClosed method **523**
reduce method **537**
sorted method **539**
sum method **523**, 536
summaryStatistics method **536**
toList method **544**

IntSummaryStatistics class **536**
IntToDoubleFunction functional interface 701
IntToLongFunction functional interface 701
IntUnaryOperator functional interface 541, 701
InvalidationListener interface (JavaFX) 637
inventory control 31
Invideo AI video generator 821
IoT (Internet of Things) 31
IP address **19**, 20
is-a relationship **319**

is read by a module **873**
`isAbsolute` method of `Path` **412**
`isBlank` method of class `String` **216**
`isCancelled` method of class `Task` **749**
`isDefined` method of class `Character` **234**
`isDigit` method of `Character` **234**
`isDirectory` method of `Files` **412**
`isEmpty` method

- `ArrayList` 297
- `Map` **489**
- `PriorityQueue` **481**
- `String` **67**

`isInterrupted` method of class `Thread` **708**
`isJavaIdentifierPart` method of class `Character` **234**
`isJavaIdentifierStart` method of class `Character` **234**
`isLetter` method of class `Character` **234**
`isLetterOrDigit` method of class `Character` **234**
`isLowerCase` method of class `Character` **234**
`isNaN` method of `Double` **441**
ISO 3166-1 Alpha-2 Country Codes 167
ISO 639-1 Language Codes 167
`isPresent` method of `Optional` **551**
`isUpperCase` method of class `Character` **234**
`Iterable<T>` interface 356
`iterate` method of interface `IntStream` **563**
iteration **81**, 928

- counter controlled 85, 88
- definite **79**
- `do...while` statement 109
- of a `for` loop 182
- of a loop 78, **100**, 117
- sentinel controlled 83, 84, 85
- statement **72**
- `while` statement 101

iteration statement 71, **72**, 72, 77, 928

`do...while` 72
`for` 72
`while` 72, **77**, 81

iteration terminates 78
iterative (non-recursive) 921
iterative factorial solution 927
iterator 436, **456**

fail fast **462**

`Iterator` interface **457**

`hasNext` method **461**
`next` method **462**
`remove` method **462**

`iterator` method of `Collection` **461**

J

Jackson open-source library xxxviii, xli, xlix, 406, **419**, 975

`@JsonIgnoreProperties` annotation **428**, 428
CSV processing 434, 436

Jacopini, G. 71
jagged array 190
Jakarta EE (formerly Java Enterprise Edition) 4
JAR (Java archive) file 881
`jar` command **883**

`--main-class` option **883**

Java 2, **155**
Java 23 lxii
Java API 154

`String` class 67

Java API (Application Programming Interface) 2, **11**, **48**, 57, 154

documentation 59

Java Application Programming Interface (Java API) 57
Java archive (JAR) file 881
Java class library **11**
Java Collection Framework **199**

interfaces 358, 457

`java` command **14**

`--describe-module` option **877**, 882
`-Duser.country` option **167**
`-Duser.language` option **166**
`-Duser.timezone` option **167**
`--enable-preview` option **724**
`--module` option **882**
`--module-path` option **882**
run a multiple-source-file program 18
run a single-source-file program 18
`--show-module-resolution` option **904**

Java compiler **13**
Java Concurrency API 700
Java Concurrency Package **155**
Java Cryptography Architecture (JCA) **252**
Java Cryptography Extensions (JCE) **252**
Java Date/Time API li, 132, **161**
Java Development Kit (JDK) lxii, 12, 52, 873
Java Enterprise Edition (Java EE) **4**
`.java` extension **12**, 264
Java Flight Recorder 755
Java Input/Output Package **155**
`java` interpreter 53
Java Language Package **155**
Java Language Specification 61
Java Math Package **155**
Java Micro Edition (Java ME) **4**
Java Platform 873

Java Platform Module System (JPMS) **868**
Java Regular Expressions Package **155**
Java Runtime Environment (JRE) 873
Java runtime options **166**
Java SE 8 199

`Arrays` method `parallelSort` 199
`default` interface methods 357
`default` methods in interfaces **357**
lambda **358**
`static` interface methods 358

Java SE 9

JShell enter an expression 953
JShell feedback mode 980
JShell recall a previous snippet 951
JShell snippet ID **952**
JShell view method documentation 966
JShell view method parameters 966

Java SE Platform 869, 873
Java Security Package **155**
Java Standard Edition (Java SE) **4**
Java Text Package **155**
Java Utilities Package **155**
Java Virtual Machine (JVM) 3, **13**, 873
`JAVA_HOME` environment variable 902
`java.base` module **156**, 872, **872**, 877, 880

contents 877

`java.io` package 155, **408**
`java.lang` package **58**, 67, 152, 155, 208, 322, 332, 707
`java.math` package 86, **125**, 155, 590

`BigInteger` class **95**

`java.net.http` package xlix, 406, **430**
`java.nio.file` package **408**, 409, 558
`java.se` aggregator module **891**

`java.security` package **142**, 252, 254
`java.sql`
    module 877
    package 155, 843, 845
`java.sql` package 155
`java.text` package 155, **580**, 590
`java.time` package 155, **161**, 162
    `Duration` class **162**
    `LocalDate` class **161**
    `LocalDateTime` class **161**
    `LocalTime` class **161**
    `Period` class **162**
`java.time.format` package **162**
    `DateTimeFormatter` class **162**, 166
    `FormatStyle enum` 162
    `TextStyle enum` 162
`java.time.temporal` package **162**
    `ChronoUnit enum` 162
`java.util` package **57**, 155, 200, 254
`java.util.concurrent` package 155, 700, 707, 717, 734, 740
`java.util.function` package 155, 540
`java.util.random` package **137**
`java.util.regex` package 155, 209, **246**
`java.util.stream` package 155, **522**
`javac` command
    `-d` option 881
    `--enable-preview` option **724**
    `-source` option **724**
`javac` compiler **13**, 53
Javadoc comment 49
    `javadoc` utility program **49**

JavaFX **568**, 640

`@FXML` annotation 591, **591**
`Accordion` class 599
`ActionEvent` class 591, 592
alignment in a `VBox` **573**
`AnchorPane` class 599
`AnchorPane` layout 649
animation xlii
`Application` class **579**, 605, 613
application thread 741, 742, 743, 745, 747
`Arc` class **649**, 651
`ArcType enum` **689**
`AudioClip` class **662**
`BorderPane` class 598, 599, **601**
`ButtonBar` class 599
`CallBack` interface **624**
`Canvas` class **649**
Cascading Style Sheets (CSS) 569
`ChangeListener` interface **581**, 593
`Circle` class **649**, 651
controller class **571**, 581, 587, 605
ControlsFX open source project 663
CSS property name 642
CSS transitions **680**
custom controls 632
`CycleMethod enum` **688**
`DialogPane` class 599
`DoubleProperty` class 593
`Ellipse` class **649**, 651
`EventHandler<ActionEvent>` interface **592**
`FadeTransition` **673**
`FillTransition` **672**
`FlowPane` class 599
fx:id property **582**
`-fx-arc-height` CSS property **653**, 687
`-fx-arc-width` CSS property **653**, 687
`FXCollections` class **617**

-fx-fill CSS property **653**
-fx-font CSS property **643**
-fx-font-family CSS property **643**
-fx-font-size CSS property **643**, 644
-fx-font-style CSS property **643**
-fx-font-weight CSS property **643**
FXGL (game library) **694**
FXML (FX Markup Language) **569**
FXMLLoader class **581**, 589
-fx-padding CSS property **642**
-fx-rotate CSS property **682**
-fx-spacing CSS property **642**
-fx-strikethrough CSS property **644**
-fx-stroke CSS property **652**
-fx-stroke-line-cap CSS property **653**, 687
-fx-stroke-width CSS property **652**
-fx-underline CSS property **644**
game development xlii, 694
graphics xlii
GridPane class **579**, 579, 580, 599
GUI xlii
HBox class 599
ID selector **643**, 644
Image class **621**
ImageView class **571**
Interpolator class **673**
InvalidationListener interface 637
javafx.scene.control.skin package **633**
KeyFrame animation class **675**
Label class **571**, 641
layout pane **599**
Line class **649**, 651
ListCell class **622**
ListView class 598
maximum size 599
MaxWidth property 585, **599**
Media class **662**, 665, 666

MediaPlayer class 662, 665, 667, 668
MediaView class 662, 665, 667
MinHeight property 599
minimum size 599
MinWidth property 599
modularized 870
MultipleSelectionModel class 621
node 570
Node class 570
node events 602
observable property 611
ObservableList 608, 617, 846, 846
ObservableValue interface 593
onMouseClicked event 602
onMouseDragEntered event 602
onMouseDragExited event 602
onMouseDragged event 602, 602
onMouseDragOver event 602
onMouseDragReleased event 602
onMouseEntered event 602
onMouseExited event 602
onMouseMoved event 602
onMousePressed event 602
onMouseReleased event 602
padding 586
Padding property 586, 603
Pane class 598, 599, 602
ParallelTransition 672
Parent class 581, 589
Path class 655
PathElement class 655
PathTransition 673
Polygon class 655
Polyline shape 655
Pref Height property 574, 585
Pref Width property 574, 585
preferred size 573, 599

prefHeight property 599
prefWidth property 599
property binding 611
Property interface 611
property listener 611
RadioButton class 598
RadioButton control 601
RadioButtonMenuItem class 608
ReadOnlyObjectProperty class 621
Rectangle class 598, 608, 649, 651
register event handlers 590
Rotate class 660
RotateTransition 673
Scale class 660
ScaleTransition 674
scene 570
Scene class 570, 579, 589, 590
scene graph 570
ScrollPane class 599
SelectionModel class 621
SequentialTransition 674
Shape class 649
Shape3D class 649
Slider class 570, 577, 580, 580, 598
SplitPane class 599
stage 570
Stage class 570, 579, 589, 590
StringBinding class 616
StringProperty class 616
StrokeLineCap enum 687
StrokeTransition 672
style class selector 644
Subscription API 633
Tab class 599
TableView class 631
TableView control 846
TabPane class 599

TextField class 570, 580
TextFlow class 600
TilePane class 600
Timeline animation class 675
TitledPane class 598, 600, 601, 604
ToggleButton class 608
ToggleGroup 601
ToolBar class 600
Transform class 660
Translate class 660
TreeTableView class 631
VBox layout 573, 600, 641
video xlii

JavaFX application thread 741
JavaFX Scene Builder 569, 571
JavaFX shapes

Circle 601
Rectangle 598

JavaFX Software Development Kit (SDK) lxv, 568
javafx.application package 579
javafx.beans.binding package 668
javafx.beans.property package 611
javafx.beans.value package 593
javafx.collections package 617
javafx.concurrent package 741
javafx.event package 591, 592
javafx.fxml package 591
javafx.scene package 570, 581, 589
javafx.scene.canvas package 649, 935
javafx.scene.control package 580, 591
javafx.scene.control.skin package 633
javafx.scene.controls package 599
javafx.scene.image package 685
javafx.scene.input package (JavaFX) 608
javafx.scene.layout package 573, 579, 599

`javafx.scene.media` package **662**
`javafx.scene.paint` package 685
`javafx.scene.shape` package **601**, 649
`javafx.stage` package 570, 625
`javafx.util` package **624**
JavaScript Object Notation (JSON) xlix, 406
`javax.crypto` package 155, **252**, 254
JCP (Java Community Process) **8**
JDBC lii

    API **830**, 864
    driver **830**

JDBC Package **155**
`jdeps` command **896**
JDK 52
JDK (Java Development Kit) 12
JEP (JDK Enhancement Proposal) **8**
JEP 200

    The Modular JDK 870, 871

JEP 201

    Modular Source Code 871

JEP 220

    Modular Run-Time Images 871

JEP 260

    Encapsulate Most Internal APIs 871, 894

JEP 261

    Module System 871

JEP 275

    Modular Java Application Packaging 871

JEP 282

jlink: The Java Linker 871

JEP 396

Strongly Encapsulate JDK Internals by Default 871

JEP 403

Strongly Encapsulate JDK Internals by Default 871

`jlink` (the Java linker) 872
`jlink` command **902**
`join` method

of `CompletableFuture` 797

`join` method of class `StructuredTaskScope` **728**
joining 835
joining database tables **835**, 839
`joining` method of class `Collectors` **534**
`jpackage` tool for packaging Java apps 917
JShell xlii, **946**, 948

`/set editor` command **977**
`CLASSPATH` 975
command **947**
compilation errors 950
continuation prompt `...>` **955**
declare a variable 950
default value of an uninitialized variable 950
enter an expression 953
execute a statement 949
execute the last valid snippet 953
external snippet editor 982
feedback mode **955**, 980
forward referencing an undeclared method **972**
implicit variable **953**
`import` types 974
input types 949
instant feedback liii
Java code snippets **949**

`jshell` command to launch JShell **948**
`jshell>` prompt **948**
list of valid snippets 952
recall a previous snippet 951
replace a declaration 950
reset a session 955
shortcut key-combinations 979
snippet ID **952**
view class documentation 966
view field documentation 966
view method documentation 966
view method parameters 966
view overloaded methods 966
viewing declared types 961
viewing declared variables 962

JShell (Java SE 9)

feedback mode 980
recall a previous snippet 951

`jshell` command **948**
JShell commands

`/!` **953**, 978
`/?` **978**, 978
`/drop` 978
`/edit` **956**, 978, 979
`/env` 978
`/env -add-modules` **975**
`/env -class-path` **975**
`/env -module-path` **975**
`/exit` **959**, 978
`/help` **978**, 978
`/history` **951**, 978
/id **952**, 978
`/imports` **974**, 978
`/list` **952**, 978
`/methods` **972**, 978

/open **963**, 978
/reload 978, **979**
/reset **955**, 978
/save **963**, 978
/set 978, 982
/set start 974
/types **961**, 978
/vars **962**, 978
drop **980**
set feedback **980**
set start 979, **982**

JShell Edit Pad 956, 957
jshell> prompt **948**
JSON (JavaScript Object Notation) xli, **418**

array 418, 419
Boolean values 419
data-interchange format 418
false 419
JSON object 418
null 419
number 419
serialization **419**
string 419
true 419

@JsonIgnoreProperties annotation **428**, 428
JsonSchema class **805**
jsonSchema method of Response-Format **805**
JsonSchemaBuilder class **805**

name method **805**
schemaClass method **805**

JSR 277: Java Module System 869
JSR 376: Java Platform Module System 869, 870, 871
JSR 379: Java SE 9 871
JSRs (Java Specification Requests) **8**

just-in-time (JIT)
    compilation 15
    compiler 15
JVM (Java Virtual Machine) 13

K

key in a search 199
key points (extracting) 768
key size in AES 257
key–value pair 482
keyboard 568
`KeyFrame` class (JavaFX) 675
`KeyGenerator` class
    `generateKey` method 257
    `getInstance` method 257
    `init` method 257
`KeyPair` class 447
    `getPublic` method 447
`KeyPairGenerator` class
    `generateKeyPair` method 447
    `getInstance` method 447
    `initialize` method 447
`keySet` method
    of class `HashMap` 488
keywords 50, 72
    `abstract` 337
    `break` 114
    `case` 114
    `catch` 382
    `char` 58

class **50**, 264
contextual 50
continue **117**
default **114**, 115
do 72, **109**
double **58**, 83, 134
else 72
enum **145**
extends **322**, 326
false **73**, 74
final 116, **154**, 176, 732
finally 382, **388**
float **58**, 83
for 72, **101**
if 72
implements **350**
import **57**
instanceof **346**
int **58**
interface **351**
module **873**, 880
new 58, 173, 210
non-sealed **367**, 368
null **172**, 271
package 798, 879
private **265**, 286, 297
public **50**, 134, 264, 286
record **310**
requires 880
reserved but not used 50
return **135**, 137
sealed 50
static **97**
super **329**
switch 72
this **265**, 287, 304
true **73**, 74

`try` **381**
`var` **135**
`void` **51**, 134
`while` 72, **109**

Koch Curve fractal **931**
Koch Snowflake fractal **932**
Koenig, Andrew 376

L

`L` for `long` integer literal **97**
`Label` class (JavaFX) **570**, 641

`Font` property 574
`Text` property 574

label in a `switch` **114**
lambda 616
lambda expression **358**, 526

anonymous method **526**
arrow token (`->`) **526**
event handler 569, 581, 616
method references 527
parameter list 526
statement block 526
target type **541**
type inference 541
with an empty parameter list 527

language codes

ISO 639-1 167

language identification **239**
language package 155
large language model (LLM) **32**, **37**, 209, 237, 249

GPT (Generative Pre-Trained Transformer) 38

last-in, first-out (LIFO) order 510
`last` method of `SortedSet` **486**
last-in, first-out (LIFO) **156**
`lastIndexOf` method of class `String` **218**
late binding **345**
launch a virtual thread 713
`launch` method of `Application` (JavaFX) **579**, 589
law of large numbers **139**
layout containers (JavaFX) **571**
layout pane (JavaFX)

- `getChildren` method **608**

Layouts

- `GridPane` **579**, 579, 580
- `Vbox` **573**

`layoutX` property of a `Node` **651**, 651
`layoutY` property of a `Node` **651**, 651
lazy evaluation **527**
lazy quantifier **244**
ledger (blockchain) **24**
left align output **107**
left brace `{` **51**, 51
left justification 1002
legacy code 7
lemmatization **240**
`length`

- instance variable of an array **172**
- method of class `String` **67**, 211
- method of class `StringBuilder` **226**
- property of an `Arc` **651**

level of indentation 74
lexical scope 542
lexicographical comparison 67, **213**, 213, 215, 250
Library window in Scene Builder 573, 574
LIFO (last-in, first-out) 156, 510

`LIKE` operator (SQL) 837, **837**
limerick 815
`limit` method of interface `IntStream` **563**
`LIMIT` SQL keyword **834**, 836
`Line` class (JavaFX) **649**, 651

- `endX` property **651**
- `endY` property **651**
- `startX` property **651**
- `startY` property **651**

linear gradient **654**
`LINEAR` interpolator **673**
linear run time 942
linear search 203
`linear-gradient` CSS function **654**, 689
line-continuation character 881
`lines` method of `Files` **558**
`lineSeparator` method of class `System` **229**, 474
`LineTo` element of a `Path` **674**
`LinkedBlockingQueue` 740
`LinkedList` class **459**

- `add` method **468**
- `addFirst` method **468**
- `addLast` method **468**

linker 872
Linux operating system 52
`List` interface **414**, **457**, 459, 466, 469, 470, 475

- `add` method **460**, 463
- `addAll` method **464**
- `clear` method **466**
- `get` method **461**
- `listIterator` method **465**
- `of` method **414**, **490**, 549
- `size` method **461**, 466
- `stream` method **545**

subList method 465
toArray method 467

/list JShell command 952
list valid snippets in JShell 952
ListCell class (JavaFX) 622

setGraphic method 624
updateItem method 622

ListIterator interface 459

hasPrevious method 466
previous method 466
set method 465

listIterator method of interface List 465
--list-modules option of the java command 870
ListView class (JavaFX) 598

custom cell factory 598
getSelectionModel 621
setCellFactory method 622, 624
setItems method 620

literal

character or digit 241
floating point 83

literal value

binary 1023
hexadecimal 1023
octal 1023

live-code approach xxvii, xxx
livelock 756
Llama (Meta) 759, 813
load factor 482
load method of class FXMLLoader (JavaFX) 581, 589
load method of class ServiceLoader 909

loading **14**
local resolution operator in FXML 648
local variable **80**, 80, 134, 147, 148, 265
local-variable type inference 135
`LocalDate` class **161**
    `getDayOfWeek` method **164**
    `parse` method **163**
    `until` method **165**, 166
`LocalDateTime` class **161**
    `format` method **166**
    `now` method **163**
    `of` method **163**
    `toLocalDate` method **164**
locale **161**
`Locale` class
    `getDefault` method **165**
locale-specific formatting **161**, 161
`LocalTime` class (`java.time` package) **161**
    `parse` method **163**
local-variable type inference 314
location-based services 31
`log` method of `Math` 153
logarithm 153
logic error **13**, 59, 76, 102, 105, 141
logical AND (`&&`) operator 120
logical complement operator `!` **121**
logical negation `!` **121**
logical operators **118**, 121
`Long` class **97**, 458
    `MAX_VALUE` constant **96**, 97
`long` primitive type 50, **95**, 111

`L` for literals **97**

`long` promotions 160
`longs` method of class `SecureRandom` **561**
`LongStream` interface **534**
look-and-feel 648
loop 81

body 109
continuation condition **72**, 102
counter 100
statement **72**

loop-continuation condition **100**, 100, 109, 110, 117
looping **81**
loosely coupled **360**, 872, 905

system 879

lossless compression **786**
lower bound of a wildcard **516**
lowercase letter 50

M

m-by-*n* array **189**
machine learning xlix, **34**, 407, 432
macOS 52
main class in a modular JAR file 889
`main` method

instance `main` 51

main thread **711**
`--main-class` option to `jar` **883**
make your point (game of craps) 142
malware detection 31
Mandelbrot, Benoit 931
many-to-many relationship **836**, 836
many-to-one mapping **486**

map elements of a stream 526
Map interface 457, 486
    containsKey method 488
    entry method 491
    forEach method 555
    get method 488
    isEmpty method 489
    nested interface Entry 491
    of method 491
    ofEntries method 491
    put method 488
    size method 489
map method of interface IntStream 526
map method of interface Stream 547
Map.Entry interface 560
    nested in Map 491
mapMulti method of Stream 566
mapping (streams) intermediate operation 525
mapping elements of a collection 465
MappingIterator class 436
    readAll method 436
mapToDouble method of interface Stream 556
mapToObj method of interface Int-Stream 533, 535
Margin property of a VBox 603
MariaDB ColumnStore 830
markdown (text-based formatting language) 770, 776
    Getting Started guide 770
    overview of markdown syntax 770
marker interface 350
marketing
    analytics 31

mashup **21**, 424
`Matcher` class 209, **246**

`find` method **246**, 248
`group` method **247**, 248

`matcher` method of class `Pattern` **246**
`matches` method of class `String` **241**
matching `catch` block 382
matching patterns in text 240
`Math` class **108**, 152, 153

`abs` method 153
`ceil` method 153
`cos` method 153
`E` constant **153**
`exp` method 153
`floor` method 153
`log` method 153
`max` method 153
`min` method 153
`PI` constant **153**
`pow` method **108**, 108, 152, 153
`sin` method 153
`sqrt` method 152, 153, 159
`tan` method 153

Maven build tool 888
Max Height property of a column 605
Max Height property of a `Pane` 604
Max Height property of a `VBox` 604
`max` method

`Collections` 469, **475**
`IntStream` **536**
`Math` 153

Max property

`Slider` (JavaFX) 587

Button 604
column 605
JavaFX control 585
Pane 604

MAX_VALUE constant

Long class **96**, 97

maximize a window 580
maximum 441
maximum size (JavaFX) 599
MaxWidth property of JavaFX controls **599**
measures of central tendency **442**
Media class (JavaFX) **662**, 665, 666
median 441, **442**
MediaPlayer class (JavaFX) **662**, 665, 667, 668

pause method **668**
play method **668**
seek method **668**
setOnEndOfMedia method **667**
setOnError method **668**
setOnReady method **668**

MediaView class (JavaFX) **662**, 665, 667

setMediaPlayer method **667**

medical device monitoring 31
MEDIUM constant of the FormatStyle enum 166
megabytes (MB) 28
memoization 927
memory 5

utilization 483

memory leak 303, 388
memory-space/execution-time tradeoff 482

merge records from tables 839
`message` method of `ChatRequest-Builder` **800**
Meta 759, 813
Meta Movie Gen video generator 32, 821
metacharacter (regular expressions) **242**
metadata **845**
metaverse xlii, 22
method **9**, 51

- argument **52**, 67, 135
- call **9**, 135
- call stack **156**
- camel case naming 134
- declaration 135
- header **134**
- local variable **134**, 265
- overloading **150**
- parameter **134**
- parameter list **134**, 193
- signature **151**
- `static` 97

method call 135
method names

- came case naming 264

method reference 527, **532**, 533

- bound instance method **533**
- constructor reference **559**
- `static` method **533**
- unbound instance method **547**

methods implicitly `final` 348
`/methods` JShell command **972**, 978
microservices **26**
Microsoft

- SQL Server 830

Microsoft Azure 20
Microsoft Copilot 239
Microsoft Windows 113
Midjourney image generator 32
migration 872
`min` method of `Collections` 469, **475**
`min` method of interface `IntStream` **536**
`min` method of `Math` 153
Min Width property of a column 604
`MinHeight` property of JavaFX controls **599**
minimize a window 580
minimum 441
minimum size (JavaFX) 599
minus sign (`-`) formatting flag **107**
`MinWidth` property of JavaFX controls **599**
missing data **438**
mixed reality xlii, **23**
mobile application 3
modal dialog 629, 852
modal window 957
model (in MVC architecture) 581
`model` method of `ChatRequest-Builder` **800**
`model` method of `ImageRequest-Builder` **809**
`model` method of `ModerationRequestBuilder` **811**
`model` method of `SpeechRequest-Builder` **807**
`model` method of `TranscriptionRequestBuilder` **806**
Model-View-Controller (MVC) 581
moderation API (OpenAI) **794**, 811

    "omni-moderation-latest" model **794**

`ModerationRequest` class **811**

    `builder` method **811**

`ModerationRequestBuilder` class **811**

    `input` method **811**
    `model` method **811**

`Moderations` class

`create` method **812**

`moderations` method of `SimpleOpenAI` **812**
modular JAR file **883**, 884, 889

main class 889

module 156, **869**, 904

is read by a module **873**
reads a module **873**
root folder **877**

module artifact **881**

locations 881

module body **873**
`module` contextual keyword **873**
module declaration **869**, 871, 873, 876, 877, 880, 887
module dependency **873**

graph **880**, 891

module descriptor **873**, 886
module directive **873**

`opens` 873, **875**
`opens...to` **875**, 900
`provides...with` 873, **874**, 913
`requires` 873, **873**, 873, 880, 887
`requires static` 873
`requires transitive` 873
`uses` 873, **874**, 911

`module` keyword 50, **880**
module name **873**, 880

`--module` option of the `java` command **882**

module path **881**, 882, 883, 886

`--module-path` option of `javac` 578
module resolver 872, 884, **904**
module-dependency graph 872, 880
`module-info.class` **873**
`module-info.java` **873**
`--module-path` option of the `java` command **882**
monetary calculations 109
    `BigDecimal` class **124**, 125
Moore's Law **5**, 5
mount a virtual thread **714**
mouse 568
`MouseEvent` class (JavaFX) **608**
move a mouse 580
`MoveTo` element of a `Path` **658**
MP3 audio format 785
multi-core processor 5, 521
multi-`catch` **382**
multi-core performance xlii
multidimensional array 189, 190
multiline statements in JShell 955
multiline string (text block) 52, **113**
multimodal genAI **758**, 759
multiplayer capabilities 694
multiple class declarations
    in one source-code file 287, 288
multiple inheritance 321
multiple-selection statement **72**
`MultipleSelectionModel` class (JavaFX) **621**
    `selectedItemProperty` method **621**
multiplication
    `*` **60**, 61
    compound assignment operator, `*=` 91
`multiply` method of class `BigDecimal` **127**

`multiply` method of the `BigInteger` class **97**
multi-release JAR file 917
multithreading 6, **699**
Murati, Mira xxviii
mutable data **732**
mutable reduction operations **544**
mutator method **296**
mutual exclusion **732**
mutually exclusive options 601
MVC (Model-View-Controller) 581
MySQL database 830

N

`n` conversion character **1006**
`%n` format specifier (line separator) **56**
`n` method of `ImageRequestBuilder` **809**
`name` method of `JsonSchemaBuilder` **805**
name of a variable **58**
named constant **145**, 176
named entity recognition (NER) xxxviii, **239**, 764, 780, **780**, 804
named module 890
naming convention
GUI (Graphical User Interface) component 582
`NaN` (not a number) **379**
`nanoTime` method of class `System` 679
`native` keyword 50
natural comparison method **469**
natural language **238**
natural language processing (NLP) xlix, 31, **41**, 208, 237, **239**, 240, 407
natural language translation 775
natural language understanding (NLU) 38, **41**, 239
natural logarithm 153
natural order 547

`negate` method

functional interface `IntPredicate` **553**
functional interface `Predicate` **553**

nested array initializers 190
nested class 263, 801

`static` 801

nested control statements **87**, 140

Examination-results problem 88

nested `for` statement 179, 191, 192, 193
nested `if`...`else` selection statement **74**
nested parentheses **61**
nested `StructuredTaskScope` 731
nested type **607**
Netflix Open Source Software Center 27
network message arrival 377
neural network 34
`new` keyword 50, **58**, 173, 210
`new Scanner(System.in)` expression 58
`newCachedThreadPool` method of class `Executors` **709**
`newDirectoryStream` method of class `Files` **412**
`newHttpClient` method of `HttpClient` **430**
`newInstance` method of class `ScopedValue` **726**
newline character **54**
newline escape sequence `\n` 55
`newVirtualThreadPerTaskExecutor` method of class `Executors` **713**
`next` method

of `Iterator` **462**
of `ResultSet` **845**
of `Scanner` **232**, 267

`nextBigDecimal` method of class `Scanner` **277**
`nextDouble` method of class `Scanner` **135**
`nextInt` method of `RandomGenerator` **138**

`nextLine` method of class `Scanner` **163**, 267
n-grams **240**
NLP (natural language processing) 31, 208, **239**
no-argument constructor 211, **290**, 295
no-op (no operation) 289
`Node` class (JavaFX) **570**, 593, 602

`layoutX` property **651**, 651
`layoutY` property **651**, 651
`requestFocus` method **593**

node in a JavaFX application **570**
`non-sealed` keyword **367**, 368
non-daemon thread 714
non-deterministic random numbers **142**
nondeterministic random numbers xlvi, **142**
nonfatal

logic error **76**
runtime error **15**

nonfungible token (NFT) xlii, 25
`non-sealed` keyword 50
NoSQL database

MariaDB ColumnStore 830

Notepad 12
notes (in the UML) **72**
`notify` method of `Object` 332
`notifyAll` method of `Object` 332
noun phrase extraction **240**
`now` method of class `LocalDateTime` **163**
`null` in JSON 419
`null` keyword **172**, 267, 271, **271**
`NullPointerException` exception 282
`Number` class **458**, 513

`doubleValue` method **514**

number systems 235

decimal (base 10) **1018**
hexadecimal (base 16) **1018**
octal (base 8) **1018**
positional notation **1019**
positional value **1019**
symbol values **1019**

`NumberFormat` class **580**, 590, 703

`format` method **593**, 703
`getCurrencyInstance` method **591**
`getPercentInstance` method **591**, 703
`setRoundingMode` method 593

numbers in JSON 419
numeric classes 458
NW 541

O

O(1) **942**
O(*n*) 942
O(*n*2) 943
object

of a class 8, 10

`Object` class 320, **322**

`clone` method 332
`equals` method 332
`getClass` method 332, **346**
`hashCode` method 332, 483
`notify` method 332
`notifyAll` method 332
`toString` method 324, 332
`wait` method 332

object of a derived class 333
object oriented

analysis and design (OOAD) **11**
programming (OOP) **11**
object-oriented programming (OOP) xlv, 4, 318
`ObjectMapper` class **420**
`readValue` method **424**
`writerWithDefaultPrettyPrinter` method **976**
`writeValue` method **421**
objects-natural approach xxvii, 3, 48, 95, 162, 199, 252, 48
Objects-Natural Case Study 65, 86
Creating and Using Objects of Standard-Library Class string 65
Intro to Regular Expressions 240
Introduction to Collections and Class ArrayList 199
Java Date/Time API 161
Precise Monetary Calculations with Java API Class `BigDecimal` 124
Super-Sized Integers 94
Symmetric-Key Cryptography 252
`ObjectWriter` class **434**
`writeValue` method **434**
`writeValueAsString` method **977**
observable module **880**, 904
observable property (JavaFX) 611
`observableArrayList` method of class `FXCollections` **620**
`ObservableList` interface (JavaFX) **608**, 617, 846, **846**
`ObservableValue` interface (JavaFX) **593**, 616
`addListener` method **593**
octal (base 8) number system **1018**
octal integer 1003
octal literal value 1023
`of` method
of `IntStream` **535**

of `List` **414**, 490, 549
of `LocalDateTime` **163**
of `Map` **491**
of `Path` 411
of `Set` **491**

`ofEntries` method of interface `Map` **491**
off-by-one error **102**
`offer` method of `PriorityQueue` **480**
`ofLocalizedDateTime method of class DateTimeFormatter` **166**
`ofPlatform` method of class `Thread` **716**
`ofString` method of `HttpResponse.BodyHandlers` **430**
`ofVirtual` method of class `Thread` **716**
On Action property of a `RadioButton` 608, 609
`ON` clause **839**, 839
On Mouse Dragged property of an `Pane` 608
`ONE` constant of class `BigDecimal` **127**
`ONE` constant of class `BigInteger` 925
one-to-many relationship **835**, 835
one-to-one mapping **486**
one's complement **1025**
ones position 1019
online forums lv
`onMouseClicked` event (JavaFX) **602**
`onMouseDragEntered` event (JavaFX) **602**
`onMouseDragExited` event (JavaFX) **602**
`onMouseDragged` event (JavaFX) 602, **602**
`onMouseDragOver` event (JavaFX) **602**
`onMouseDragReleased` event (JavaFX) **602**
`onMouseEntered` event (JavaFX) **602**
`onMouseExited` event (JavaFX) **602**
`onMouseMoved` event (JavaFX) **602**
`onMousePressed` event (JavaFX) **602**
`onMouseReleased` event (JavaFX) **602**
OOAD (object-oriented analysis and design) 11
OOP (object-oriented programming) **11**, 318
opacity 611

open a file **408**
`/open` JShell command **963**, 978
`open` keyword **875**
open source
    code liv
    software liv
Open Web Application Security Project (OWASP) 419
OpenAI 36
    API key 762
    ChatGPT 36
    Dall-E API **786**
    developer account 761
    prompt engineering guide 40
OpenAI API 760
    advanced reasoning models 779
    Audio **806**, 807
    audio **784**
    audio formats, MP3 785
    audio generation 761
    chat completion 799, **799**
    image generation 761
    model 799
    moderation 761, **794**, 811
    `OPENAI_API_KEY` 799
    reasoning 761
    speech to text 761
    structured output 764, 780
    text generation 760
    text to speech 761
    Text to Speech documentation 784
    TTS-1 model xxxviii
    vector embedding 761
    vision 761, 802
OpenAI chat completion

"assistant" role 767
"system" role 767
"user" role 767

OpenAI ChatGPT lxvii, 239
OpenAI Dall-E image generator **32**
OpenAI models 760

DALL-E 760
"dall-e-3" model **786**, 786
GPT-4o 760
GPT-4o Audio 760
GPT-4o mini 760
GPT-4o-mini 760
GPT-4o-mini Audio 760
reasoning 760
TTS (text-to-speech) 760
"tts-1-hd" **784**
Whisper (speech-to-text) 760
Whisper model **783**
"gpt-4o" **767**

OpenAI Sora video generator 32, **792**, 821
`OPENAI_API_KEY` 799
`OpenAIUtilities` class 763, 797

`chat` method 799

`opens` module directive 50, 873, **875**
`opens...to` module directive **875**, 900
`openStream` method of `URL` **809**
OpenWeatherMap xli, 406, 452

Current Weather Data 425
One Call API 425

OpenWeatherMap web service 424
operand **59**, 85
operating system thread 710
operation in the UML **269**

operation parameter in the UML 269
operator **60**
operator precedence 61, 926

rules **61**

operators

-- predecrement/postdecrement **91**
-- prefix decrement/postfix decrement 92
! logical NOT 119, **121**
?: ternary conditional operator 77
*= multiplication assignment operator 91
/= (division assignment) 91
/= division assignment operator 91
& boolean logical AND 118, **121**
&& conditional AND **119**, 120
%= remainder assignment operator 91
^ boolean logical exclusive OR 119, **121**
++ prefix increment/postfix increment 92
++ preincrement/postincrement 91
+= addition assignment operator **91**
= **60**
-= subtraction assignment operator 91
-= (subtraction assignment) 91
| boolean logical inclusive OR 119, **121**
|| conditional OR 118, **120**
arithmetic **60**
binary **59**, 60
boolean logical AND & 118, **121**
boolean logical exclusive OR ^ 119, **121**
boolean logical inclusive OR | **121**
cast **85**
compound assignment 91, 93
conditional AND && **119**, 121
conditional operator ?: **77**
conditional OR || 118, **120**, 121
decrement operator -- 91, 92

dot (.) **67**
logical AND, `&&` 120
logical complement `!` **121**
logical negation `!` **121**
logical operators **118**, 121, 122
multiplication `*` **60**
pattern-matching `instanceof` 365
postfix decrement **92**
postfix increment **92**
prefix decrement **92**
prefix increment **92**
remainder `%` **60**, 61
subtraction `-` 61

optimizing compiler 108
`Optional` class 551

`ifPresent` method **551**
`isPresent` method **551**

optional dependency 873
`OptionalDouble` class **536**, 557

`getAsDouble` method **536**, 557
`orElse` method **536**

`OptionalInt` class **536**

`getAsInt` method **536**

`or` method of functional interface `Int-Predicate` **553**
`or` method of functional interface `Predicate` **553**
Oracle Corporation 830
order 70
`ORDER BY` SQL clause 836, **838**, 838
ordering of records 836
O'Reilly Online Learning free trial lx
`orElse` method of class `OptionalDouble` **536**
out-of-bounds array index 377
outer set of brackets 183

outliers 241
output 52
output cursor 52, 54
output parameter for a `CallableStatement` **864**
overflow 377
overload

- constructors **289**
- generic methods 505
- method **150**, 496

override a superclass method **322**, 324
OWASP (Open Web Application Security Project) 419

P

`\p{Punct}` punctuation character class **250**
package **57**, 154

- access **309**, 309
- name 270

package-access methods 370
`package` keyword 50
`package` statement 798, 879
Packages 599

- default package **270**
- `java.io` 155, **408**
- `java.lang` **58**, 67, 152, 155, 208, 322, 332, 707
- `java.math` 86, **125**, 155, 590
- `java.net.http` 406, **430**
- `java.nio.file` **408**, 409, 558
- `java.security` **142**, 252, 254
- `java.sql` 155, 843, 845
- `java.text` 155, **580**, 590
- `java.time` 155
- `java.util` **57**, 155, 200, 254
- `java.util.concurrent` 155, 700, 707, 717, 734, 740

java.util.function 540
java.util.random **137**
java.util.regex 155, 209, **246**
java.util.stream **522**
javafx.application **579**
javafx.beans.binding 668
javafx.beans.property **611**
javafx.beans.value 593
javafx.concurrent **741**
javafx.event 591, **592**
javafx.fxml 591
javafx.scene 570, **581**, 589
javafx.scene.canvas **649**, 935
javafx.scene.control 580, 591
javafx.scene.control.skin **633**
javafx.scene.image 685
javafx.scene.layout 573, **579**
javafx.scene.media **662**
javafx.scene.paint 685
javafx.scene.shape 649
javafx.stage 570, 625
javax.crypto **252**, 254

packaging Java apps

jpackage tool 917

packets (on the Internet) **19**
padding (JavaFX) **586**
Padding property of a JavaFX layout container 586, 603
Paint class **686**, 687, 688, 689
pair of braces {} 64
Pane class (JavaFX) 598, 599, **602**

Max Height property 604
Max Width property 604
On Mouse Dragged property 608
Pref Height property 604
Pref Width property 604

parallel **698**
`parallel` method of a stream 705
parallel operations 698
parallel stream 705
parallel stream operations xlix
parallelism xlii
`parallelSetAll` method of class `Arrays` **701**
`parallelSort` method of class `Arrays` 199, 701
`ParallelTransition` class **672**
parameter **134**
parameter in the UML 269
parameter list **134**
parameter list in a lambda 526
parameterized type **505**
`Parent` class (JavaFX) **581**, 589
parent node 599
parentheses **51**, 61
    nested **61**
`parse`
    method of class `LocalTime` **163**
`parse` method
    of class `LocalDate` **163**
`parseInt` method of class `Integer` 458
`parseInt` method of `Integer` 196
parsing text 241
particle effects 694
parts-of-speech (POS) tagging **240**, **782**, 823
pass an array element to a method 186
pass an array to a method 186
pass-by-reference **188**
pass-by-value 186, **188**
passing options to a program 194
`Path` class (JavaFX) **655**

ArcTo element 658
ClosePath element 659
LineTo element 674
MoveTo element 658
PathElement 658

Path interface 409

getFileName method 411
isAbsolute method 412
of method 411
toAbsolutePath method 412
toString method 412

PATH_TO_FX environment variable lxvi, 578

macOS lxvi
Windows lxvi

PathElement class (JavaFX) 655, 658
path-separator character 904
PathTransition class 673
Pattern class 209, 246

CASE_INSENSITIVE constant 247
compile method 246
matcher method 246
splitAsStream method 559

pattern matching 837, 837

decomposing record object 314
in text 240
instanceof 314, 346, 365
switch 314
switch expressions 313
unnamed variables (_) 401

pattern-matching switch 367
pause method of class MediaPlayer 668
Payable interface declaration 351

`Payable` interface test program processing `Invoice`s and `Employee`s polymorphically 355
`peek` method of `PriorityQueue` **480**
percent (`%`) SQL wildcard character 837
perform a task 135
performing operations concurrently **698**
`Period` class

- `between` method **164**
- `getDays` method **165**
- `getMonths` method **165**
- `getYears` method **165**
- `toTotalMonths` method **165**

`Period` class (`java.time` package) **162**
`permits` clause of a `sealed` class or interface **367**
`permits` keyword 50
Perplexity lxvii, 36
personalized medicine 31
personally identifiable information (PII)

- removing 249

petabyte (PB) 29
petaflops 6
phishing elimination 31
phong shading 692
`PhongMaterial` class **692**

- `setDiffuseColor` method **693**
- `setDiffuseMap` method **693**
- `setSpecularColor` method **693**
- `setSpecularPower` method **693**

physics engine 694
plaintext in cryptography 253
`Platform` class **741**

- `runLater` **741**

platform dependent **13**

platform independent **13**
platform thread 700, **710**, 712
    non-daemon thread by default 715
`play` method of class `MediaPlayer` **668**
`play` method of class `Transition` **674**
player input handling 694
PNG image format **786**
`PointLight` class 691
`poll` method of `PriorityQueue` **480**
pollution reduction 31
`Polygon` class **655**
`Polyline` class (JavaFX) **655**
polymorphic processing of collections 458
polymorphic processing of related exceptions 385
polymorphic video game 332
polymorphically process `Invoice`s and `Employee`s 355
polymorphism 116, **319**, 333
pop off a stack **156**
portable **13**
position number 172
positional notation in number systems **1019**
positional value 1019
positional value in number systems **1019**
positional values in the decimal number system 1019
postcondition **397**
postfix decrement operator **92**
postfix increment operator **92**
PostgreSQL 830
postincrement 92
post-quantum cryptography 7
`pow` method
    class `BigDecimal` **127**
    class `Math` **108**, 108, 152, 153
power (exponent) 153
prebuilt data structures 456

pre-built generic data structures xlix
precedence **61**, 926
    chart 62
precise monetary calculations 124
    `BigDecimal` class **124**, 125
precision 1002, 1003, 1004
    `BigDecimal` 125
    format of a floating-point number **86**, 86
precision medicine 31
precondition **397**
predicate 529
`Predicate` functional interface 541, 544, 547, 550
    `and` method **553**
    `negate` method **553**
    `or` method **553**
predicate method **297**
Pref Height property
    `GridPane` 603
    JavaFX control 574, 585
    `Pane` 604
    `VBox` 604
Pref Width property
    column 604
    `GridPane` 603
    JavaFX control 574, 585
    `Pane` 604
    `VBox` 604
preferred size (JavaFX) **573**, 599
`prefHeight` property of a node **599**
prefix decrement operator **92**

prefix increment operator **92**
`prefWidth` property of a node **599**
preincrement 92, 93
`PreparedStatement` interface **852**, 853, 854, 855, 864

- `executeQuery` method **856**
- `executeUpdate` method **857**
- `setString` method **853**, 856

`prepareStatement` method of interface `Connection` **855**
preparing data for analysis 209
press a mouse button 580
preventable coding errors 387
preventative medicine 31
preventing disease outbreaks 31
`previous` method of `ListIterator` **466**
primary key **831**, 831, 833, 834, 835

- composite 835

primitive type **59**, 160, 171

- `boolean` **73**
- `byte` 111
- `char` 58, 111
- `double` **58**, 83
- `float` **58**, 83
- `int` 58, 60, 111, 959
- `long` 95, 111
- passed by value 189
- `short` 111

principal in an interest calculation 106
principle of least privilege **149**, 308, 348
`print` method of `System.out` **54**
print on multiple lines 54
print spooling **733**
`printf` method of `System.out` **56**, 1002
`println` method of `System.out` 54
`printStackTrace` method of class `Throwable` **392**

`PriorityQueue` class **480**
`clear` method **480**
`isEmpty` method **481**
`offer` method **480**
`peek` method **480**
`poll` method **480**
`size` method **480**

privacy xli, xliv, 94, 252
`private`

access modifier **265**, 286, 287, 370
data 297
keyword 50, 297

`private` interface methods (Java SE 9) **358**
private key, RSA **446**
private-key cryptography 155, **253**, 407
`private static`

class member 304

probability 138
procedural programming xlv, 4
processing steps in a stream pipeline 523
producer 699, **733**
producer thread **733**
producer/consumer relationship **733**
program execution stack **156**
program in the general 319
program in the specific 319
programming paradigms 520

concurrent 4
concurrent programming xlv
functional 4, 520
functional-style xlv
genAI enhanced programming xlv
generative-AI-assisted 4
generic xlv, 4, 520

object-oriented xlv, 4, 520
procedural xlv, 4, 520

programming to an interface not an implementation 905
programming to an interface, not an implementation xli
`progress` property of class `Task` **747**, 749
Project Jigsaw **868**
Project Loom xlii, li, 700, **712**, 723

scoped value xlii
structured concurrency xlii
virtual thread xlii

promotion **85**

of arguments **159**
rules 86, **160**

prompt engineering
prompt engineering (genAI) xxxiv, xxxvi, **39**

guide 40
sample prompts 40

prompt message **58**

`prompt` method of `ImageRequest-Builder` **809**

property **361**
property binding (JavaFX) 611, 616

bidirectional 616

property injection **361**
`Property` interface (JavaFX) **611**
property listener (JavaFX) 611, 616
property of an object **611**
`protected` access modifier 50, 286, 370
protecting the environment 31
protocol for communication (`jdbc`) **844**
`provides` keyword 50
`provides...with` module directive 873, **874**, 913

`public`

- `abstract` method 352
- access modifier 134, 264, 286, 370
- `final static` data 352
- interface **283**
- keyword **50**, 50
- member of a subclass 371
- method 284, 287
- method encapsulated in an object 286
- service **283**
- `static` class members 304
- `static` method 304

public encryption key **446**
public key

- RSA **446**

public-key cryptography xli, 407, 452
`public` keyword 890
public-key cryptography 95, 155, **259**, **445**
punctuation

- removing 249

punctuation character class

- `\p{Punct}` **250**

pure function in functional programming **540**
push onto a stack **156**
`put` method

- of interface `BlockingQueue` **734**, 736
- of interface `Map` **488**

Q

quadratic run time **943**
qualified export **874**, 879, 900

`qualified exports...to` module directive 879
qualified name **839**, 839
`quality` method of `ImageRequest-Builder` **809**
quantifier
`?` **244**
`{n,}` **245**
`{n,m}` **245**
`*` **243**
`+` **244**
greedy **244**
in regular expressions **243**
lazy **244**
quantum computers 6
query **830**, 831
a database 842
query method **296**
queue 480, 733
`Queue` interface **457**, 480
QuickTime 791

R

`radial-gradient` CSS function **653**
radians 153
`RadioButton` class (JavaFX) 598
On Action property 608, 609
`setToggleGroup` method 601
Text property 604
Toggle Group property 604
`RadioButton` control (JavaFX) **601**
`RadioButtonMenuItem` class (JavaFX) 608
`radius` property of a `Circle` **651**
`radiusX` property of a `Arc` **651**

`radiusX` property of a `Ellipse` **651**
`radiusY` property of a `Arc` **651**
`radiusY` property of a `Ellipse` **651**
radix (base) of a number **235**
random numbers
    deterministic **142**
    difference between values 142
    element of chance **137**
    generation 278
    nondeterministic **142**
    scaling factor 142
    secure 142
    shifting value 142
random sequence reproducible **474**
`RandomGenerator` 137
    `getDefault` method **137**
    `nextInt` method **138**
`RandomGenerator` interface
    streams of random numbers 532, 561
random-number generation
    13 algorithms 137
range checking **82**
`range` method of class `EnumSet` **303**
`range` method of interface `IntStream` **523**
range-view methods **465**, 484
`rangeClosed` method of interface `IntStream` **523**
ratio of successive Fibonacci numbers 924
reactive programming 723
read a module **873**, 886
readability 49, **239**
`readAll` method of `MappingIterator` **436**
`readAllBytes` method of `Input-Stream` **808**
`readerFor` method of `CsvMapper` **436**

read-evaluate-print loop (REPL) liii, **946**
reading files 409
`ReadOnlyObjectProperty` class 668
`ReadOnlyObjectProperty` class (JavaFX) **621**
`readString` method of `Files` **766**
`readValue` method
    of `ObjectMapper` **424**
`readValues` method of `CsvMapper` **436**
real number 58
realization in the UML **351**
recall a previous snippet in JShell 951
reclaim memory 306
recommender systems 31
record 413
`record` class **310**, 619, 620
    constructor 311
    decompose in a pattern match **314**
    `equals` method 312
    get method 312
    keyword **310**
    patterns **314**
`Rectangle` class (JavaFX) 598, **649**, 651
    `height` property **651**
    `width` property **651**
recursion
    memoization 927
    overhead 928
    recursion step **920**, 925
    recursive backtracking **941**
    recursive call **920**, 925, 926
    recursive `factorial` method 922
    recursively generating Fibonacci numbers 926
recursive method **920**

redirect a standard stream **408**
`reduce` method of interface `DoubleStream` 557
`reduce` method of interface `Int-Stream` **537**
reducing carbon emissions 31
reducing energy use 31
reduction in streams **523** reduction operations 528
refactoring **26**, 511, 894

payroll example 359

reference **171**

type **171**, 171

reference type 309, 370
referential transparency **540**
reflection 869, 872, 874, 900
regex (regular expression) 208, 240
`regionMatches` method of class `String` **215**
register event handlers (JavaFX) **581**, 590
regular expression 208, **240**, 558

`?` quantifier **244**
`[] character class` **243**
`{n,}` quantifier **245**
`{n,m}` quantifier **245**
`*` quantifier **243**
`\` metacharacter **242**
`\d` character clas 242
`\D` character class **242**
`\d` character class **243**
`\p{Punct}` punctuation character class **250**
`\S` character class **242**
`\s` character class **242**
`\W` character class **242**
`\w` character class **242**
`+` quantifier **244**
caret (`^`) metacharacter **243**
case insensitive 241

case sensitive 241
character class **242**, 243
escape sequence **243**
metacharacter **242**
`Pattern.CASE_INSENSITIVE` **247**
search pattern 240
validating data 240

reinforcement learning 34, **35**
reinventing the wheel 57, 124, **154**, 197
relational database **830**, 831

relational database management system (RDBMS) **830**, 830

relational operators **62**
relative path **409**
relative positioning 599
release a resource 388
reliable configuration 869, 884, 887, 907
`/reload` JShell command 978
remainder 61

compound assignment operator, `%=` 91
operator, `%` **60**, 61

remove duplicate `String` 483
`remove` method of class `Array-List<E>` 200, **203**
`remove` method of interface `Iterator` **462**
`removeAll` method of `Collection` **462**
`removeFirst` method of `ArrayList<E>` 200
`removeLast` method of `ArrayList<E>` 200
`removeLast` method of class `Array-List` 507
rendering 694
Reordering output with argument index 1013
`repeat` method of class `String` **179**
repetition

`do...while` statement 110
sentinel controlled 85

REPL (read-eval-print-loop)

see JShell **946**, 946, 948

REPL (read-evaluate-print-loop) xlii
`replace` method of `String` **222**
`replaceAll` method of `String` **245**
`replaceFirst` method of class `String` **246**
replaces a declaration in JShell 950
representational error 124, 125

in floating point **108**

Representational State Transfer (REST) **425**
representing integers in hexadecimal 1002
representing integers in octal 1002
reproducibility **253**
reproducible random sequence **474**
request to a web service **424**
`requestFocus` method of class `Node` **593**, 593
requirements **11**
`requires` keyword 50
`requires` module directive 873, **873**, 873, 880, 887
`requires static` module directive 873
`requires transitive` module directive 873
reserved word `null` 267
reset a JShell session 955
`/reset` JShell command **955**
resizable array implementation of a `List` 459
resolver **904**
resource encapsulation 898
resource leak 303, **388**
resource-release code 388
resources 898
response from a web service **424**
`ResponseFormat` class **805**

`jsonSchema` method **805**

`responseFormat` method of `ChatRequestBuilder` **805**

responseFormat method of TranscriptionRequestBuilder 810
responses to a survey 181
RESTful web services 425
restricted keywords 876
ResultSet interface 844
    close method 845
    column name 845
    column number 845
    cursor 845
    getInt method 845
    getObject method 845
    next method 845
ResultSetMetaData interface 845
    getColumnCount method 845
    getColumnType method 845
resumption model of exception handling 383
rethrow an exception 390
return keyword 50, 135, 137
return statement 920
return type
    of a method 134
reusable software components 8, 505
reuse 9, 57, 67
reverse domain name convention 877
reverse method of class StringBuilder 227
reverse method of Collections 468, 475
reversed method of interface Comparator 547
reverseOrder method of Collections 470
rgb method of class Color 616
RGBA color model 611
rgba CSS function 653
ride sharing 31
right align 1007

right align output **107**
right brace (}) **51**, 51, 80, 85
right justification 1002
right justifying integers 1008
right-align the contents of a column 585
robo advisers 31
robust application 59, 376
roll back a transaction **864**
`rollback` method of interface `Connection` **865**
rolling two dice 144
root directory **409**
root folder of a module **877**
root module 905
root node **570**
`Rotate` class **660**
`rotate` method of class `Transform` **662**
`RotateTransition` class **673**
round a floating-point number for display purposes 86
round half to even **127**
rounding 61, 1002
  a number 81, 109, 153
  banker's **127**
  fair rounding algorithm 124
  half even **127**
  half up **127**
  policy of a `BigDecimal` **125**, 125
  policy `RoundingMode.HALF_EVEN` **127**
  round half to even **127**
`RoundingMode` enum 590
`RoundingMode.HALF_EVEN` rounding policy **127**
row in a database table **831**, 831, 835, 836, 837, 839
Row Span property of a control in a `GridPane` 613
rows of a two-dimensional array 189
RSA cryptography xli
RSA public-key cryptography 407, **446**
RSA Public-Key Cryptography algorithm **95**

Rule of Entity Integrity **835**, 835
Rule of Referential Integrity **835**, 835
rules of operator precedence **61**, 926
run a multiple-source-file program with the `java` command 18
run a single-source-file program with the `java` command 18
run an application from a module 876
`run` method of class `ScopedValue.Carrier` **726**
`run` method of interface `Runnable` **707**
`runLater` method of class `Platform` **741**
`Runnable` interface 357, **667**, **707**

`run` method **707**

runtime annotation 428
runtime error **15**
runtime polymorphism **347**
`RuntimeException` class **386**
Runway Gen-3 Alpha video generator 32

S

`SalariedEmployee` concrete class extends `abstract` class `Employee` 341
SAM (single abstract method) interface 358, 540
sample prompts for genAI 40
`/save` JShell command **963**, 978
savings account 106
scalable Java Platform 869
scalar **186**
`Scale` class **660**
`ScaleTransition` class **674**
scaling (random numbers) scale factor 142
`Scanner` class 57, **58**

`close` method **417**
`hasNext` method **114**
`next` method **232**, 267

`nextBigDecimal` method **277**
`nextDouble` method **135**
`nextLine` method **163**, 267

Scene Builder 568, **569**, 571

content panel 571
Controller 587, 588, 605
GUI development tool lxvii
Hierarchy 575, 603
Inspector 573
Library 573, 574
select multiple controls 587
Show Preview in Window option 571, 576

`Scene` class (JavaFX) **570**, 579, 589, 590
scene graph 581, 590, 598, 599
scene graph in a JavaFX application **570**
scene in a JavaFX application **570**
scene transitions 694
scheduled state of a `Task` 746
`schemaClass` method of `JsonSchemaBuilder` **805**
`schemaFor` method of `ScvSchema` **434**
science, technology, engineering and math (STEM) xxx
scientific notation 1004
scope

of a declaration **147**
of a variable **103**
structured concurrency 700

scope of a type parameter 507
scoped value xlii, lii, 700
scoped values **723**
`ScopedValue` class 724, **726**

`get` method **727**
`newInstance` method **726**
`where` method **726**

`ScopedValue.Carrier` class **726**

`run` method **726**

scraping 241
screen cursor 55
screen-manager program 333
`ScrollPane` class (JavaFX) 599
SDK (Software Development Kit) **26**
`sealed`

class xli, **367**
interface xli, **367**
keyword 50
`permits` clause **367**

search key **199**
search pattern (regular expressions) 240
secondary storage 5

device **406**

secret key 253
secret-key cryptography **253**, 259, xli
`SecretKey` class 257, 258

`getEncoded` method **255**

`SecretKey` interface 451
`SecretKeySpec` class **451**
secure random numbers 142
`SecureRandom` class **142**, 142, 257

`doubles` method **561**
`ints` method **532**, 561
`longs` method **561**

security xli, 14, 94, 252
security breach 142
Security Developer's Guide 260
`SecurityException` class **415**
seed a random number generator for reproducibility 253

`seek` method of class `MediaPlayer` **668**
select multiple controls (Scene Builder) 587
`SELECT` SQL keyword **834**, 834
`selectAll` method of class `TextInputControl` **592**, 593
`selectDouble` method of class `Bindings` **668**
`selectedItemProperty` method of class `MultipleSelectionModel` **621**
selecting an item from a menu 580
selection 72
selection criteria **837**, 837
selection statement 71, **72**

- `if` 72, 73
- `if...else` 72, **74**, 74
- `switch` 72, **111**, 115

`SelectionModel` class (JavaFX) **621**
selector 642
self-contained Java applications 917
self-driving cars 31, 32
self-similar property **931**
self-attention (generative AI) **38**
semicolon (`;`) **52**, 58, 64
send a message to an object 9
sentiment analysis 31, **239**, 768
sentinel-controlled iteration 84, 85
sentinel value **83**, 85
separator character **412**
sequence 72, **459**
sequence structure **71**
sequential execution **70**
sequential file 406, 413, 452
sequential text file 406
`SequentialTransition` class **674**
serialization **350**, 406, **419**

- avoid language native serialization 419
- CSV 434
- pure data formats 419

security 419

service 874
service consumer **911**
service of a class 287
service provider 872, **874**, 906, 913
`ServiceLoader` class 905, **909**

`load` method **909**
`stream` method **909**

service-provider interface 879, 906, 907
services 872
`/set editor` command in JShell **977**
`Set` interface **457**, 483, 484, 486

`of` method **491**
`stream` method 560

`/set` JShell command 978
`/set start` JShell command 974
`set` method of interface `ListIterator` **465**
`SET` SQL clause **840**, 840
`setAutoCommit` method of interface `Connection` **865**
`setCellFactory` method of class `ListView` **622**, 624
`setCellValueFactory` method of `TableColumn` **852**
`setCharAt` method of class `StringBuilder` **227**
`setDiffuseColor` method of class `PhongMaterial` **693**
`setDiffuseMap` method of class `PhongMaterial` **693**
`setErr` method of class `System` **408**
`setFill` method of class `GraphicsContext` **687**
`setGlobalAlpha` method of class `GraphicsContext` **687**
`setGraphic` method of class `ListCell` **624**
`setImage` method of class `ImageView` **621**
`setIn` method of class `System` **408**
`setItems` method of class `ListView` **620**
`setItems` method of `TableView` **852**
`setLength` method of class `StringBuilder` **226**
`setLineWidth` method of class `GraphicsContext` **686**

`setMediaPlayer` method of class `MediaView` **667**
`setOnEndOfMedia` method of class `MediaPlayer` **667**
`setOnError` method of class `MediaPlayer` **668**
`setOnReady` method of class `MediaPlayer` **668**
`setOnRunning` method of class `Task` **746**
`setOnSucceeded` method of class `Task` **746**
`setOut` method of `System` **408**
`setRoundingMode` method of class `NumberFormat` 593
`setScale` method of class `BigDecimal` **127**
`setSpecularColor` method of class `PhongMaterial` **693**
`setSpecularPower` method of class `PhongMaterial` **693**
`setString` method of interface `PreparedStatement` **853**, 856
`setStroke` method of class `Graph-icsContext` **686**
`setStroke` method of class `Graph-icsContext` class `setStroke` method **940**
`setText` method of class `TextInput-Control` **592**, 593
`setUserData` method of a JavaFX control **602**, 657
shadow a field **148**, 265
shallow copy **332**
`Shape` class (JavaFX) **649**
`Shape` class hierarchy 321
`Shape3D` class (JavaFX) **649**, 690
share memory 699
shared buffer 734
shared immutable data **732**
shared mutable data **732**
shell **52**
shift (random numbers) shifting value 142
short-circuit evaluation **120**
`Short` class **458**
`short` primitive type 50, 111

promotions 160

short-circuting terminal operation 551
shortcut key combinations (JShell) 979
Show Preview in Window option (Scene Builder) 571, 576
`showDialog` method of class `DirectoryChooser` **630**

-show-module-resolution option to java 904
showOpenDialog method of class FileChooser 629
showSaveDialog method of class FileChooser 629
shuffle 278
    algorithm 472
shuffle method of class Collections 469, 472, 474
shuffling
    Fisher-Yates algorithm 281
shutdown method of class Executor-Service 715
shutdown method of interface ExecutorService 710
side effect 121
Sieve of Eratosthenes 749
signal value 83
signature of a method 151, 152
significant digits of a BigDecimal 125
silicon 4
similarity detection 31, 239
simple condition 118
SimpleOpenAI class 799
    audios method 806
    builder method 799
    chatCompletions method 800
    images method 809
    moderations method 812
Simple-OpenAI library 763, 797
    AssistantMessage 800, 801
    Audios class 806
    Base64Util class 803
    ChatCompletions class 800
    ChatMessage 801
    ChatMessage>default para font> class 800
    ChatRequest class 800, 803
    ChatRequestBuilder class 800

ContentPart class 803
ContentPart nested class ContentPartImageUrl 803
ContentPart nested class ContentPartText 803
DeveloperMessage class 800
ImageRequest class 809
ImageRequestBuilder class 809
Images class 809
JsonSchema class 805
JsonSchemaBuilder class 805
ModerationRequest class 811
ModerationRequestBuilder class 811
Moderations class 812
ResponseFormat class 805
SimpleOpenAI class 799
SimpleOpenAIBuilder class 799
SpeechRequest class 807
SpeechRequestBuilder class 807
SystemMessage 799, 801, 803
SystemMessage>default para font> class 800
TranscriptionRequest class 806
TranscriptionRequestBuilder class 806
UserMessage 800, 801, 803

Simple-OpenAI open-source library lii
SimpleOpenAIBuilder class 799

apiKey method 799
build method 799

simulate a coin toss 138
simulate a rolling a six-sided die 138
simulation 137
sin method of class Math 153
sine 153
single abstract method (SAM) interface 540
single-entry/single-exit control statements 73
single inheritance 321
single-line (end-of-line) comment 52

single-precision floating-point number **83**
single-quote character 209
single-selection statement **72**, 73
single-element collection 493
single-selection `if` statement 73
SingleStaticImportDeclaration **307**
`singleton` method of class `Collections` 493
`singletonList` method of class `Collections` 493
`singletonMap` method of class `Collections` 493
`size` method
    of class `ArrayBlockingQueue` **736**
    of class `ArrayList<E>` **203**
    of class `PriorityQueue` **480**
    of interface `List` **461**, 466
    of interface `Map` **489**
`size` method of class `Files` **412**
skinning **569**, 633
`sleep` method of class `Thread` **708**, 737, 738, 739
    Block Increment property 587
    Max property 587
    Value property 587
    `valueProperty` method 593, 616
small circles in the UML **71**
smart contracts 24
smart homes 31
smart thermostats 31
smart traffic control 31
smartmeters 31
smartphone 3
snippet ID (JShell) **952**
snippets of Java code in JShell **949**
Software Development Kit (SDK) **26**
software development technologies xlii
software engineering 297

reuse 67

software reuse **12**, 136
solid circle

in the UML **72**
surrounded by a hollow circle in the UML **72**

Sora AI video generator 760
Sora video generator 32, **792**, 821
`sort` method

of class `Arrays` **197**, 198
of class `Collections` **469**

`sort` method of class `Arrays` 539, 701
`sorted` method of interface `IntStream` **539**
`sorted` method of interface `Stream` **544**, 545, 547
sorted order 484, 486
`SortedMap` interface **486**
`SortedSet` interface **484**, 486

`first` method **486**
`last` method **486**

sorting

descending order 470
with a `Comparator` 471

sound effects 694
`-source` option (`javac` and `java` commands) **724**
source code 12
source language in translation 764
space character 50
space flag 1010
Spacing property of a `VBox` 603, 604
spam detection 31
`speak` method of `Audios` **807**
special character 58, **209**
specialization **319**

speech 789
    recognition (speech-to-text) 783
    synthesis (text-to-speech) 783, 784
speech recognition **239**
speech synthesis **239**
`SpeechRequest` class **807**
    `builder` method **807**
`SpeechRequestBuilder` class **807**
    `input` method **807**
    `model` method **807**
    `voice` method **807**
spell checking **240**
spelling correction **240**
`Sphere` class **690**
spiral 924
`split` method of class `String` **237**
`splitAsStream` method of class `Pattern` **559**
`SplitPane` class (JavaFX) 599
`SpotLight` class 691
SQL (Structured Query Language) **830**, 831, 836, 840, 852
    `DELETE FROM` statement 836, **841**
    `FROM` clause 836
    `GROUP BY` clause 836
    `INNER JOIN` clause **839**
    `INSERT INTO` statement 836, **839**
    keyword 836
    `LIMIT` keyword **834**, 836
    `ON` clause **839**
    `ORDER BY` clause 836, **838**, 838
    percent (`%`) wildcard character 837
    `SELECT` query **834**
    `SET` clause **840**
    `UPDATE` statement 836

VALUES clause **840**
WHERE clause **837**

SQL injection attack (preventing) 853
SQL statement 865
SQLException class **844**, 845, 853
SQLite 831

changes function **840**

SQLite database management system xlii, 830, 832
sqlite3 command (to create a database) **833**, 854
sqrt method of class Math 152, 153, 159
square brackets [] **172**
Stable Diffusion image generator 32
stack **156**, 505

frame **156**, 928
overflow **157**, 923
trace **379**, 974
unwinding **391**

stack chunk in virtual threads **712**
Stack generic class 505

declaration 506
Stack<Double> 512
Stack<Integer> 512

StackOverflow (stackoverflow.com) liv, lvi
StackPane 599
StackTraceElement class **393**

getClassName method **393**
getFileName method **393**
getLineNumber method **393**
getMethodName method **393**

StackWalker class **393**, 565
Stage class (JavaFX) **570**, 579, 589
stage in a JavaFX application **570**

stale value **732**
standard error stream (`System.err`) **382**, 408
standard input stream (`System.in`) **58**, 408
standard module **870**, 877, 880, 888
standard output stream (`System.out`) **52**, 408
standard time format 284
`start` method of class `Application` (JavaFX) **579**, 589
`start` method of class
`Thread.Builder.OfVirtual` **716**
start tag **646**
`startAngle` property of a `Circle` **651**
`startsWith` method of class `String` 68, 216
`startVirtualThread` method of class `Thread` **716**
`startX` property of a `Line` **651**
`startY` property of a `Line` **651**
state dependent **733**
statement **52**, 135
`break` **114**, 116, 117
`continue` **117**
control statement 71, 72
control-statement nesting **73**
control-statement stacking **73**
`do...while` 72, **109**
double selection **72**
empty 64, 76
enhanced `for` **184**
`for` 72, **101**, 108
`if` **62**, 72, 73
`if...else` 72, **74**, 74
iteration 71, **72**, 77
looping **72**
multiple selection **72**
nested **87**
nested `if...else` **74**
`return` 137

selection 71, **72**
single selection **72**
`switch` 72, **111**, 115
`throw` **284**
`try` **183**
`try`-with-resources **399**, 414, 417
`while` 72, 77, 81, 101

statement block in a lambda 526
`Statement` interface **844**, 845, 852

`close` method 845
`executeQuery` method **844**

`static`

class member 304
class variable 304
field (class variable) **303**
import **307**
keyword 50

method 97
static binding **348**
`static` interface method **358**
`static` interface methods 358
`static` method in an interface 540
`static` method reference **533**
`static` nested class 801
StaticImportOnDemandDeclaration **307**
statistics

average 441, **442**
descriptive 441, 442
maximum 441
measures of central tendency **442**
median 441, **442**
minimum 441

status code of an `HttpResponse` 430

steganography 819
stemming **240**
stock market forecasting 31
`Stop` class **688**
stop word elimination **240**
stored procedure **864**
straight-line form **61**
stream **523**
  `DoubleStream` interface **534**
  filter elements 529
  intermediate operation **525**, 527
  `IntStream` interface **522**, 534
  `LongStream` interface **534**
  map elements to new values 526
  mapping intermediate operation **525**
  parallel 705
  pipeline **523**
  pipeline processing steps 523
  reduction **523**
  terminal operation **524**
`Stream` interface **542**
  `collect` method **544**, 555, 556
  `distinct` method **554**
  `filter` method 544, 545, 547
  `findFirst` method 551
  `flatMap` method **559**
  `map` method **547**, 547
  `sorted` method **544**, 545, 547
  `toArray` method **544**
`stream` method
  class `Arrays` **543**
  class `ServiceLoader` **909**
  interface `List` **545**
  interface `Set` 560

`stream` method of class `Arrays` 543
stream of bytes **407**
streaming 699
`strictfp` keyword 50
strictly self-similar fractal **931**, 936
string **51**
    literal **51**
    of characters 51
string built-in type
    in JSON 419
`String` class **65**, 67, 208
    `charAt` method **211**
    `compareTo` method **215**
    `concat` method **221**
    constructor that copies another `String`'s contents **211**
    constructor that receives a `char` array **211**
    constructor that receives a `char` array and two integers **211**
    `endsWith` method **68**, 216
    `equals` method **67**, 213, 214
    `equalsIgnoreCase` method **214**
    `format` method **284**, **1014**
    `getBytes` method 257
    `getChars` method **211**
    `indexOf` method **218**
    `isBlank` method **216**
    `isEmpty` method **67**
    `lastIndexOf` method **218**
    `length` method **67**, 211
    `matches` method **241**
    no-argument constructor **211**
    `regionMatches` method **215**
    `repeat` method **179**
    `replace` method **222**

`replaceAll` method **245**
`replaceFirst` method **246**
searching methods 218
`split` method **237**
`startsWith` method **68**, 216
`strip` method **223**
`stripLeading` method **223**
`stripTrailing` method **223**
`substring` method **220**, 221
`toCharArray` method **223**
`toLowerCase` method **222**
`toUpperCase` 465
`toUpperCase` method **222**
`valueOf` method **223**

`String` concatenation **67**
string literal 67, **209**
`StringBinding` class (JavaFX) **616**
`StringBuilder` class 208, **225**

`append` method **228**
`capacity` method **226**
`charAt` method **227**
constructor 225
constructor that receives a capacity 225
constructor that receives a `String` 225
constructors 225
`delete` method **231**
`deleteCharAt` method **231**
`ensureCapacity` method **226**
`getChars` method **227**
`insert` method **230**
`length` method **226**
no-argument constructor 225
`reverse` method **227**
`setCharAt` method **227**
`setLength` method **226**

`StringJoiner` class **544**
`StringProperty` class (JavaFX) **616**
    `bind` method **616**
`strip` method of class `String` **223**
`stripLeading` method of class `String` **223**
`stripTrailing` method of class `String` **223**
`strokeArc` method of class `GraphicsContext` **689**
`strokeLine` method **940**
    class `GraphicsContext` **686**, 687, 940
`StrokeLineCap enum` **687**
`strokeOval` method of class `GraphicsContext` **688**
`strokeRect` method of class `GraphicsContext` **687**
`strokeRoundRect` method of class `GraphicsContext` **687**
`StrokeTransition` class **672**
strong encapsulation 869, 874, 887, **890**
Stroustrup, Bjarne 376
structure of a document 633, 641
structured concurrency xlii, lii, 700, **723**
    scope 700
structured output 764, 780
structured programming **70**, 118
Structured Query Language (SQL) xlii, **830**, 831, 836
`StructuredTaskScope` class 724, **727**
    `fork` method **727**
    `join` method **728**
    nested 731
    shuts down on failure **727**
`StructuredTaskScope.ShutdownOnFailure` class **727**
`StructuredTaskScope.Subtask` interface **728**
    `get` method **728**
style class selector (JavaFX) **642**, 644

style sheet 641
subclass **318**
sublist **465**
`subList` method of `List` **465**
`submit` method of class `Executor-Service` **717**
subprotocol for communication **844**
`Subscription` API (JavaFX) 633
substitution cipher 252
substring 218
`substring` method of class `String` **220**, 221
subtitles (closed captions) **790**
`subtract` method of class `BigInteger` 97, 925
subtraction 60

- compound assignment operator, `-=` 91
- operator `-` 61

`sum` method of interface `DoubleStream` 557
`sum` method of interface `IntStream` **523**, 536
sum an array's elements 177
summarize text 767
summarizing responses to a survey 181
summarizing text 31
`summaryStatistics` method of `IntStream` **536**
`super` keyword 50, 329
superclass **318**

- constructor 324
- constructor call syntax **329**
- default constructor 324
- direct **321**, 321
- indirect **321**, 321
- method overridden in a subclass 330

supercomputer **6**
super-sized integers 70, 94
`Supplier` functional interface 541
suppressed exception **400**
`switch` expression **141**, 146, 147, 314

arrow (->) symbol **141**, 146, 147
comma-separated list of values in a `case` 146
pattern matching 313, **314**
`yield` statement **147**

`switch` logic 116
`switch` multiple-selection statement 50, 72, **111**, 115

`case` label 114, 115
controlling expression **114**
`default` case **114**, 115

symbol values **1019**
symmetric encryption 445, **446**
symmetric-key cryptography **253**, 253
synchronization **732**
synchronize **699**
`synchronized`keyword 50
synchronous error **377**
syntax error **50**
`System` class

`arraycopy` **197**, 199
`exit` method **388**
`getenv` method **429**
`lineSeparator` method **229**, 474
`nanoTime` method 679
`setErr` method **408**
`setIn` method **408**
`setOut` **408**

`System` class's `getProperty` method 412
`system` method of interface `Instant-Source` **701**
`"system"` `role` in an OpenAI chat completion **767**
`System.err` (standard error stream) **382**, 408
`System.in` (standard input stream) 408
`System.out` **52**

`print` method **54**
`printf` method **56**

println method 52, 54
standard output stream 408

SystemMessage class 800

T

tab character \t 55, 1013
Tab class (JavaFX) 599
Tab key 51
tab stops 55
table 189

in a database 830

table element 189
table of values 189
TableColumn class setCellValue-Factory method 852
TableColumn in a TableView 851
tables 830
TableView class setItems method 852
TableView class (JavaFX) 631
TableView control (JavaFX) 846
TabPane class (JavaFX) 599
tabular format 175
tagging interface 350
tailSet method of class TreeSet 486
take method of class BlockingQueue 734, 736
tan method of class Math 153
tangent 153
target language in translation xxxvii, 764
target type of a lambda 541
Task class 741, 742

call 742, 743
cancel method 753
canceled state 746
failed state 746
isCancelled method 749

`progress` property **747**
scheduled state 746
`setOnRunning` method **746**
`setOnSucceeded` method **746**
`updateMessage` **742**, 743
`updateProgress` method **747**, 749
`updateValue` method **747**, 749
`value` property **747**, 749

TCP (Transmission Control Protocol) **19**
TCP/IP **19**
temporary 85
`TEN` constant of class `BigDecimal` **127**
terabytes (TB) 29
teraflop 6
terminal (window) **52**
terminal stream operation **524**, 526, 533, 535, 544

`average` method of interface `IntStream` **536**
`collect` method of interface `IntStream` **534**, 534, 535
`collect` method of interface `Stream` **544**, 555, 556, 562
`count` method of interface `IntStream` **536**
`findFirst` method of interface `Stream` 551
`mapToDouble` method of interface `Stream` **556**
`max` method of interface `IntStream` **536**
`min` method of interface `IntStream` **536**
mutable reduction **544**
`reduce` method of interface `Int-Stream` **537**
reduction **523**, 528
short-circuting 551
`sum` method of interface `IntStream` **523**, 536
`summaryStatistics` method of interface `IntStream` **536**
`toArray` method of interface `Stream` **544**
`toList` method of interface `Stream` **544**

termination model of exception handling **383**
termination test 928

ternary operator **77**
Test-Driven Development (TDD) **26**
text block (multiline string) 52, **113**
    format string **165**
text file 406, **408**
text generation 764
text mining 241
Text property of a `Button` 604
`Text` property of a `Label` (JavaFX) 574
Text property of a `RadioButton` 604
text summarization **239**, 764, **764**, 767
`TextArea` class **625**
TextBlob NLP library
    parts-of-speech (POS) tagging **782**
textblock
    formatted 1015
TextEdit 12
`TextField` class (JavaFX) **570**, 580
    `textProperty` method **616**
`TextFlow` class (JavaFX) 600
`TextInputControl` class (JavaFX) 592, 593
    `getText` method **592**
    `selectAll` method **592**
    `setText` method **592**
`textProperty` method of class `TextField` **616**
`TextStyle enum` (`java.time.format` package) **162**
    `FULL` **165**
text-to-speech (speech synthesis) 784
the cloud 20, 418
theft prevention 31

`thenComparing` method of functional interface `Comparator` **552**
Thinking Like a Developer liv
`this`
keyword 50, **265**, 287, 304
to call another constructor of the same class 290
thread **383**
confinement **741**
daemon 714
non-daemon 714
of execution **699**
operating system 710
safe **703**, 741
scheduling 739
synchronization **732**
`Thread` class
`currentThread` method **709**
`dumpStack` method 379
`interrupt` method **708**
`isInterrupted` method **708**
`ofPlatform` method **716**
`ofVirtual` method **716**
`sleep` method **708**
`startVirtualThread` method **716**
thread pool **707**
cached 710, 718
fixed 718
thread safe 734, 735, 741
`Thread.Builder.OfVirtual` class
`factory` method **716**
`start` method **716**
`unstarted` method **716**

`Thread.Builder.OfVirtual` interface **716**
`ThreadLocalRandom` class `current` method **703**
thread-per-request handling (virtual threads) **724**, 726
thread-per-request server code **713**
throw an exception **183**, 183, **378**, 381

`throw` statement 284, **284**, 292

`throw` keyword 50
throw point **379**, 392
`throw` statement 390
`Throwable` class **385**, 392

`getCause` method **395**
`getMessage` method **393**
`getStackTrace` method **393**
`getSurpressed` method **400**
`printStackTrace` method **392**

`throws` clause 50, **384**
tightly coupled **359**, 872
`TilePane` class (JavaFX) 600
time formatting 1002
Time Zone Codes 167
`Timeline` animation class (JavaFX) **675**
Titanic disaster dataset xli, 407, 437
`TitledPane` class (JavaFX) 598, 600, **601**, 604
`titles` table of `books` database 833
`to` keyword 50
`toAbsolutePath` method of interface `Path` **412**
`toArray` method of interface `List` **467**, 468
`toArray` method of interface `Stream` **544**
`toCharArray` method of class `String` **223**
`ToDoubleFunction` functional interface **557**
Toggle Group property of a `RadioButton` 604
`Toggle` interface (JavaFX) **609**, 609

`getUserData` method **609**

`ToggleButton` class (JavaFX) 608

ToggleGroup class (JavaFX) 601

getSelectedToggle method 608

toHours method of Duration class 165
token 237, 240
token context in genAI 39
token in generative AI 39
tokenization 237, 240
toList method of class Collectors 544
toList method of interface Stream 544
toLocalDate method of class Local-DateTime 164, 165
toLowerCase method of class Character 235
toLowerCase method of class String 222
toMillis method of class Duration 702
toMinutes method of Duration class 165
ToolBar class (JavaFX) 600, 625, 935
top-level type 607
toPath method of class File 629
Tortoise and the Hare Race 817
toSeconds method of Duration class 165
toString method 97, 184

of a collection 456
of class ArrayList 469
of class Arrays 197, 198
of class Formatter 1014
of class Object 324, 332

toString method of interface Path 412
toTotalMonths method of Period class 165
toUpperCase method of class Character 235
toUpperCase method of class String 222, 465
Towers of Hanoi 929
traditional comment 49
trailing whitespace characters 223
transaction

processing 864

transaction processing 830, 864
`transcribe` method of `Audios` 806
transcription
    audio segment 811
`Transcription` class
    `getSegments` method 811
`TranscriptionRequest` class 806
    `builder` method 806
`TranscriptionRequestBuilder` class 806
    `build` method 806
    `file` method 806
    `model` method 806
    `responseFormat` method 810
transfer of control 70
transform 660
`Transform` class 660
    `rotate` method 662
transformer (generative AI) 38
transforming data 241
    to other formats 209
`transient` keyword 50
transition arrow in the UML 71, 71, 74, 78
`Transition` class (JavaFX) `play` method 674
`transition-delay` for a JavaFX CSS transition 683
`transition-duration` for a JavaFX CSS transition 683
`transition-property` for a JavaFX CSS transition 683
`transition-timing-function` for a JavaFX CSS transition 683
transitive closure 904
`transitive` keyword 50
`Translate` class 660

translate speech 32
translation 774

    source language xxxvii, 764
    target language xxxvii, 764

Transmission Control Protocol (TCP) **19**
traverse an array **191**
tree 483
`TreeMap` class **486**, 559, 560
`TreeSet` class **483**, 484, 486

    `headSet` method **485**
    `tailSet` method **486**

`TreeTableView` class (JavaFX) 631
trend spotting 31
trigonometric

    cosine 153
    sine 153
    tangent 153

`true` reserved word **62**, 73, 74
truncate 61, **415**

    fractional part of a calculation **81**

truth tables

    for logical operators **119**
    for operator `^` 121
    for operator `||` 120

`try` block **183**, 381, 392
`try` keyword 50, **381**
`try` statement **183**, 383
`try`-with-resources statement **399**, 414, 417
tuple 310
Turing test **36**
24-hour clock format 283

`TWO` constant of class `BigDecimal` **127**
two-dimensional array **189**, 190
two's complement 1025
    notation **1025**
twos position 1020
type **58**
type argument 507
type casting **85**
type checking
    compile-time 457
type erasure **501**, 504, 507
type inference 135
type inference in generics with the `<>` notation 463
type of a variable **58**
type parameter **499**, 506, 512
    common names 500
    scope 507
    section **499**, 506
`type` property of a `Circle` **651**
type safety 498
type selector (CSS) **651**
type variable **499**
type-wrapper class 232, **458**, 502
    implements `Comparable` 502
type-parameter section 503
`TypeReference` class **424**
`Types` class **845**
`/types` JShell command **961**, 978
type-safe **200**
type-wrapper class **232**

U

UML (Unified Modeling Language) **11**

activity diagram **71**, 71, 74, 78, 110
arrow 71
class diagram **268**
compartment in a class diagram 268
diamond 73
dotted line **72**
final state **72**
guard condition **73**
merge symbol **78**
note **72**
solid circle **72**
solid circle surrounded by a hollow circle 72
`uml.org` 72

UML (`www.uml.org`) 72
unary operator 121

cast **85**

`UnaryOperator` functional interface 541
unbiased shuffling algorithm 281
unbound instance method reference **547**
unboxing a primitive value from an object 200
unboxing conversion **459**, 506, 510
uncaught exception **383**
unchecked exception **386**, 397
underlying data structure 480
underscore

_ SQL wildcard character 837
in numbers for readability 58

underscore (_) as a variable name 701
Unicode character set 94, 116, 209, 234
Unified Modeling Language (UML) **11**
Uniform Resource Identifier (URI) **430**
Uniform Resource Locator (URL) **430**
unique module name 873

universal-time format 283, 284
unmount a virtual thread **714**
unnamed module **895**, 895
unnamed variables (_) in patterns **401**
unspecified number of arguments 193
`unstarted` method of class
`Thread.Builder.OfVirtual` **716**
`UnsupportedOperationException` class **467**
`until` method of class `LocalDate` **165**, 166
unwinding the method-call stack 391
`UPDATE` SQL statement 836, **840**, 840
`updateItem` method of class `List-Cell` **622**
`updateMessage` method of class `Task` **742**, 743
`updateProgress` method of class `Task` **747**, 749
`updateValue` method of class `Task` **747**, 749
upper bound
of a wildcard 515
upper bound of a type parameter **503**, 504
uppercase letter 51
URI (Uniform Resource Identifier) **430**
`URI` class `create` method **430**
`uri` method of `HttpRequest.Builder` **430**
URL (Uniform Resource Locator) **425**, 430
`URL` class
`openStream` method **809**
`URLEncoder` class
`encode` method **429**
use cases 31
user data for a JavaFX control **602**, 657
`"user" role` in an OpenAI chat completion **767**
`uses` keyword 50
`uses` module directive 50, 873, **874**, 911
Utilities Package 155

utility method **286**

V

V's of big data 30
valid identifier 58
validate

- array index values 181
- data 82
- data (regular expressions) 240
- first name 243

validity checking 297
Value property of a `Slider` (JavaFX) 587
`value` property of class `Task` **747**, 749
`valueOf` method of class `BigInteger` **97**
`valueOf` method of class `String` **223**
`valueProperty` method of class `Slider` 593, 616
`values` method of an `enum` **302**
`VALUES` SQL clause **840**, 840
`var` keyword 50, **135**
variable 57, 58

- name **58**
- reference type **171**
- type **58**

variable declaration statement **58**
variable is not modifiable 308
variable-length argument list **193**
variable name

- camel case naming 134, 264

variable scope 103
variable-length argument list 491
variety (in big data) 31
`/vars` JShell command **962**

`VBox` class (JavaFX) **573**, 600, 641

- alignment **573**
- `Alignment` property 573, 574
- `children` element **647**
- Margin property 603
- Max Height property 604
- Pref Height property 604
- Pref Width property 604
- Spacing property 603, 604

velocity (in big data) 31
Veo video generator 32
veracity (in big data) 31
`version` attribute in `xml` declaration **646**
version control system liv, 26
version control system (VCS) **26**
version control systems (VCS) Git **26**
Vgap property of a `GridPane` 586
video xlii, 789
video game 138
video generator

- Adobe Firefly 821
- Google Veo 2 821
- Invideo AI 821
- Meta Movie Gen 32, 821
- OpenAI Sora 821

video streaming 740
view (in MVC) 581
view class documentation (JShell) 966
view field documentation (JShell) 966
view method documentation (JShell) 966
view method parameters (JShell) 966
view overloaded methods (JShell) 966
viewing declared

- types in JShell 961

variables in JShell 962

virtual machine **13**
virtual reality

devices 22

virtual reality (VR) xlii, **22**, 23

fully-immersive 22
non-immersive 22
semi-immersive 22

virtual threads xlii, li, 155, 700, 712, **712**

always a daemon thread 715
carrier **714**
`factory` method of class `Thread.Builder.OfVirtual` **716**
launch 713
mount on a platform thread **714**
`newVirtualThreadPerTaskExecutor` method of class `Executors` **713**
`ofVirtual` method of class `Thread` **716**
stack chunk **712**
`start` method of class `Thread.Builder.OfVirtual` **716**
`startVirtualThread` method of class `Thread` **716**
`Thread.Builder.OfVirtual` interface **716**
thread-per-request handling **724**, 726
unmount from a platform thread **714**
`unstarted` method of class `Thread.Builder.OfVirtual` **716**

virtualization 27
vision capabilities of the OpenAI API 802
visual product search 31
visual programming 571
Visual Studio Code lxiv, 13
visualization **177**
VLC Media Player 791, 821
`voice` method of `SpeechRequest-Builder` **807**

voice recognition 31
`void` keyword 50, **51**, 134
`volatile` keyword 50
volume (in big data) 30
VR (virtual reality) **22**
VTT (Video Text Track) format for closed captions xxxviii, **790**

W

W3C (World Wide Web Consortium) **19**
`wait` method of `Object` 332
waiting line 480
weather forecasting 31
web 424
web scraping 241
web service 20, 26, 418, 424

- client application **424**
- Free Public APIs 21
- GitHub Public APIs 21
- Google APIs Explorer 21
- host **424**
- request **424**
- response **424**
- web service host **424**

Web3 xlii, **25**
WebKit framework 634
`when` keyword 50
`when` method of class `Bindings` **668**
`where` method of class `ScopedValue` **726**
`WHERE` SQL clause 836, **837**, 837, 838, 840, 841
`while` iteration statement 50, 72, **77**, 81, 101
Whisper generative AI model (OpenAI) **783**
`"whisper-1"` audio model **783**
whitespace **50**, 51, 64

character 223, 237

widgets 568
`width` property of a `Rectangle` **651**
wildcard 515

in a generic type parameter **513**
lower bound **516**
type argument **515**, 515
upper bound 515

wildcard (`*`) in a filename **268**
William Gates liv
window gadgets 568
Windows operating system 113
`with` keyword 50
`with` method of `CsvMapper` **436**
`withHeader` method of `ScvSchema` **434**
word character **242**
word frequency counting 237, **240**
`Worker` interface **741**

`getValue` method **746**

workflow **71**
World Wide Web **19**, 19
World Wide Web Consortium (W3C) 771, 802
worst-case run time for an algorithm 942
`write` method of `FileOutputStream` **808**
write once, run anywhere (WORA) 7
`writerWithDefaultPrettyPrinter` method

of `ObjectMapper` **976**

`writeValue` method

of `ObjectMapper` **421**
of `ObjectWriter` **434**

`writeValueAsString` method

of `ObjectWriter` **977**

X

XML (eXtensible Markup Language) 419, 569
attribute 646
comment 646
declaration **646**
element **646**
empty element 647
end tag **646**
start tag **646**

Y

`yield` keyword 50
in a `switch` expression **147**

Z

z-axis 691
zero-based counting 102, **174**
`ZERO` constant of class `BigDecimal` **127**
`ZERO` constant of class `BigInteger` 925
`0` (zero) flag **1009**, 1011

Live Full-Throttle Training *with* Paul Deitel

Learn a programming language and/or key libraries in one day!

Paul's Full-Throttle training courses on O'Reilly Online Learning are fast-paced, code-intensive, presentation-only introductions to programming languages and key libraries for developers and software project managers.

- Java Full Throttle with Paul Deitel
- Python Full Throttle with Paul Deitel
- Python Data Science Full Throttle with Paul Deitel
- Modern C++ Full Throttle with Paul Deitel

See a schedule of upcoming courses and sign up at **oreilly.com/live-events**.

O'Reilly subscribers can attend Paul's live trainings and view his detailed e-book and video programming language treatments. **His 50+ hour *Java Fundamentals*, *Python Fundamentals* and *C++20 Fundamentals* videos have garnered millions of contact minutes with Java and Python, each ranking #1 for several years among thousands of video courses.**

Get your free 10-day subscription at **oreilly.com/start-trial/**.

Register Your Product at informit.com/register

Access additional benefits and save up to 65%* on your next purchase

- Automatically receive a coupon for 35% off books, eBooks, and web editions and 65% off video courses, valid for 30 days. Look for your code in your InformIT cart or the Manage Codes section of your account page.
- Download available product updates.
- Access bonus material if available.**
- Check the box to hear from us and receive exclusive offers on new editions and related products.

InformIT—The Trusted Technology Learning Source

InformIT is the online home of information technology brands at Pearson, the world's leading learning company. At informit.com, you can

- Shop our books, eBooks, and video training. Most eBooks are DRM-Free and include PDF and EPUB files.
- Take advantage of our special offers and promotions (informit.com/promotions).
- Sign up for special offers and content newsletter (informit.com/newsletters).
- Access thousands of free chapters and video lessons.
- Enjoy free ground shipping on U.S. orders.*

** Offers subject to change.*

*** Registration benefits vary by product. Benefits will be listed on your account page under Registered Products.*

Connect with InformIT—Visit informit.com/community

Pearson

informIT

Addison-Wesley • Adobe Press • Cisco Press • Microsoft Press • Oracle Press • Peachpit Press • Pearson IT Certification • Que

Reviewer Comments *(Continued from Inside Back Cover)*

■ **Carl Dea, Lead Software Developer at a global business consulting and services company, co-author *JavaFX 9 by Example***

"Great intro to **JavaFX** for career minded individuals to become the next generation of app developers. I like how the authors along with **genAI prompts** first take the reader through a design-centric approach to visualize the application's UI by using **Scene Builder** without jumping into Java source code right away.

All examples are **real-world problems.** Your SpotOn app is a **fun game with cool sound effects**. A **really fun** Chapter 16 progresses to more advanced topics including mouse events, layouts, properties, and displaying & scaling images. Many examples/exercises are reminiscent of common desktop applications such as Paint and Photo Viewer programs. Excellent use of **record classes** for a Book object.

Kudos (Loved it)!!! By far, Chapter 17 is the most compelling, fun, and comprehensive presentation of **JavaFX Graphics and Multimedia APIs** for those wanting to build **graphics apps and games**. This chapter is practically flawless!

I like how the authors begin with CSS styling on various vector-based shapes then later present how to reproduce the shapes using the **Canvas API**. Next, we learn how to transform scene graph nodes and investigate the **JavaFX animation APIs** to perform **motion effects** such as rotation, scaling & translation. The authors also demo the powerful **AnimationTimer API**.

I really like the **Media Player** example showing how to play MP4 **video** of Nasa's Artemus 1. The authors also show how to display 3D shapes and colorize their surfaces, and use cameras and light sources. A great addition is **JavaFX23 CSS transitions**. Nice to see **lambdas** improving code readability, especially in the media player event handlers. **GenAI prompts** allow the reader to gain more insight on **JavaFX** topics."

■ **José Antonio González Seco, Parliament of Andalusia, Spain**

"New in this edition is the **perfect integration of generative AI technologies** as tools to reinforce learning, deepen understanding of the topics and help in the programmer's daily work. Provides practical instruction in **prompt engineering**, which has become essential to maintaining productivity in any field. The **generative AI exercises** are very interesting and give the book a **modern feel**. It's a bit scary how well **ChatGPT** solves some of the exercises and explains its solutions. **Dependency injection (DI)** is a great addition to this fifth edition."

■ **Trisha Gee, Java Champion**

"The combination of theory, clear explanations and example code makes this both a great learning experience and an excellent reference that is useful for any developer to have close at hand. A great job covering **inheritance, polymorphism** and **interfaces**. Perfect level of **exception handling** coverage. You get a good feel for when you'd use **lambdas and streams**. Great overview of the **Java Platform Module System!** Covers all the topics I expected to see on **modularity** (more actually, especially around **visualizing dependencies**). Solid intro to **JShell** with plenty of hands-on code demos."

■ **Bob Myers, Computer Science Department, Florida State University**

"A nice mix of **core language concepts** with illustrations from many useful **Java libraries**. I like the contrast drawn in the **objects-natural** case study between use of raw floating-point types and the **BigDecimal** class type when handling **monetary amounts.**

A good overview of **class building** with solid examples. The new information about **record classes** is well-presented. A nice intro to the core concepts of **relational databases**. Great examples of core **SQL query commands** and advice on **prepared statements**.

Nice intro to tasks that AIs can accomplish. Good sections illustrating test runs for common tasks, like **translating between languages,** creating **text summaries**, **generating images**, **generating image descriptions**, etc. Lots of exercises to practice these tasks! A clear and concise discussion of **JShell**."

■ **Johan Vos, Co-founder and CTO, Cloud Products at Gluon, Java Champion, co-author *The Definitive Guide to Modern Java Clients with JavaFX: Cross-Platform Mobile and Cloud Development***

“Easy intro on how to start with **JavaFX**. It is clear that the developer doesn’t need to learn lots of new concepts/ideas, yet immediately can start creating applications with **Scene Builder**. Well done! Covers a number of fundamental **JavaFX components**, enabling developers to create real applications.”

■ **David Vlijmincx, Senior Software Developer, JPoint, co-author *Virtual Threads, Structured Concurrency, and Scoped Values: Explore Java’s New Threading Model***

“I really liked the Concurrency chapter and its breadth—it covers the basics one needs to work with platform and virtual threads.”

Generative AI Innovations in *Java for Programmers, 5e*

Fully Coded GenAI Case Studies

Chapter 19 presents the following **code examples that interact with OpenAI's APIs: Text Summarization, Sentiment Analysis, Accessible Image Descriptions, Language Detection & Translation, Java Code Generation, Named-Entity Recognition & Structured Outputs, Speech-to-Text, Text-to-Speech, Image Generation, Creating Closed Captions for a Video, Moderation**.

GenAI Prompt Exercises

We fed the complete list of all the book's **approximately 600 genAI exercises** (a 100+ page PDF) to **ChatGPT, Gemini, Claude** and **Perplexity**, asking them to categorize the kinds of things we do in those exercises. Next, we fed their categorized lists to the four genAIs, asking them to summarize the summaries, and we chose the best one—**Claude** in this case:

- **Code Generation and Implementation**—Writing new Java programs from specifications. Implementing specific features, algorithms and APIs. Creating test programs and practical applications. Generating solutions for basic and advanced tasks.
- **Code Refactoring and Enhancement**—Modernizing code. Improving code structure, readability, and maintainability. Converting between different approaches while maintaining functionality. Improving performance.
- **Educational Content**—Creating tutorials, exercises, and learning materials. Further exploring complex concepts. Developing programming exercises. Writing comprehensive documentation and guides.
- **Technical Analysis**—Analyzing code behavior and feature implementations. Comparing different approaches, tools, and

frameworks. Evaluating trade-offs in design decisions. Breaking down complex technical concepts.

- **Best Practices and Standards**—Implementing coding standards and design patterns. Addressing security considerations. Optimizing performance. Following Java development best practices.
- **Technology Evaluation**—Comparing libraries, tools, and frameworks. Assessing the pros and cons of different approaches. Making informed technology choices. Exploring new features and updates.
- **Debugging and Error Handling**—Finding and fixing syntax and logical errors. Implementing exception handling. Improving fault tolerance. Preventing common pitfalls.
- **API and Library Integration**—Working with Java APIs and external libraries. Understanding API features and capabilities. Implementing integration techniques. Creating API documentation and tutorials.
- **Real-world Applications**—Developing practical use cases and industry applications. Creating interactive applications (GUIs, games, multimedia). Implementing real-world scenarios. Building sample projects.
- **Performance Optimization**—Analyzing and improving performance. Optimizing resource usage. Conducting benchmarks. Implementing efficiency improvements.
- **Creative Development**—Building multimodal applications. Creating visualizations. Generating test scenarios and sample data. Developing unique use cases.

GenAI API-Based Java Programming Exercises

Chapter 19, Building API-Based Java Generative AI Applications, suggests challenging project exercises like creating genAI multimedia apps that can debate one another and using genAIs to build and solve crossword puzzles. We fed the 94 exercises into the genAIs, asking for a categorized summary of them, then summarized the summaries. Here's what **Claude** produced:

- **Multimodal Applications**—Building comprehensive tools that combine text, image, audio, speech and video capabilities. Creating integrated experiences like interactive books. Developing multimedia educational content.
- **Text-Based Applications**—Document processing (indexing,

summarization, exploration). Creative writing (stories, poetry, debates). Language tools (translation, tone rewriting). Professional document creation (resumes, presentations). Structured outputs.

- **Image Processing Applications**—Generative art and design (logos, fashion, floor plans). Technical visualization (UML diagrams). Image analysis and recognition.
- **Audio and Music Applications**—Speech processing (transcription, voice cloning). Music generation (MIDI, Magenta AI). Multilingual audio applications. Podcast and audio content analysis.
- **Educational Tools**—Programming tutors (Java, coding exercises). Subject-specific learning aids (math). Course content creation. Interactive educational experiences.
- **Gaming and Puzzle Applications**—Puzzle generators and solvers. Interactive game development.
- **Video**—Investigating and experimenting with generative AI video creation tools.
- **Chatbot Development**—Character-based chat experiences. Specialized domain experts.
- **Research and Analysis Tools**—Medical applications (researching drug discovery and personalized medicine). AI capability exploration. Text detection and analysis. Educational research.
- **Creative Applications**—Children's book creation. Interactive storytelling. Artistic content generation. Creative writing.
- **Practical Tools and Utilities**—Document generators. Translation services. Content summarizers. Professional tools (resume filters, presentation creators).

Reviewer Comments *(Continued from Back Cover)*

■ **Brian Canada, Professor of Computational Science, University of South Carolina Beaufort**

"Excellent coverage of the **latest Java language features.** The Strings chapter sets the stage for important applications in **NLP** (a critical ingredient to making **genAI** systems work) and **cryptography** (a critical ingredient to **information security)** that are introduced by **objects-natural** case studies in Chapters 7 **(private-key cryptography)** and 11 **(public-key cryptography).** The **objects-natural** case study on class **BigInteger** is especially pertinent at a time when **cybersecurity** is more important than ever and programmers need to develop an appreciation for **secure software engineering** practices.

Real-World Modeling with Inheritance, Polymorphism and Interfaces covers a lot but I feel smarter after reading it and going through the associated **genAI exercises!** These are cleverly written so prompted AIs will guide the reader to "fill in the gaps" and provide application areas where these techniques are used.

The **objects-natural** case study on **ArrayLists** provides a "sneak peek" at the Java Collections Framework showing how this data structure solves problems caused by the limitations of Java arrays. The **Java Collections Framework** chapter does an admirable job covering most key data structures and how **Java's built-in collections** implement them. The section on **hash tables** is informative. The coverage of newer JDK techniques, such as **convenience factory methods,** is appropriate.

The **genAI exercises** are highly valuable and fun, giving the reader new tools for filling in details and thinking critically about how and why certain practices are followed in Java in particular and object-oriented programming in general. Great job contrasting **"checked" and "unchecked" exceptions.** Suggests mechanisms for **"bulletproofing your code"** with validation logic so that certain exceptions aren't thrown in the first place. The **genAI prompt exercises** on the types of **OpenAI API** messages passed to the **chat completion API** were very helpful for cementing my understating. Amazing AI description of the beach photo! In grad school my

research group was barely scratching the surface of **identifying objects in images**—things have come a long way! The **closed captions demo** is cool!

A broad introduction to fundamental OOP **inheritance** and **polymorphism** concepts. The use of **interfaces** to implement runtime polymorphism logically follows a similar discussion involving **abstract classes.** I like the discussion of the relative benefits of **composition vs. inheritance.** Accomplishes the herculean task of covering **tight vs. loose coupling** and **dependency injection** in just a few pages. Great to see the Builder design pattern in action."

■ **Ron Veen, Java Developer, Special Agent at Team Rockstars IT, co-author *Virtual Threads, Structured Concurrency, and Scoped Values: Explore Java's New Threading Model***

"This is the best book for learning Java that I have come across in 25+ years of Java experience. I've suggested using it to train developers in my company and in the companies where I do consultancy. A detailed explanation of the core Java language. Excellent discussion of **inheritance and polymorphism.** Teaches how to use and incorporate AI in your daily software-creation process. Using tools like **ChatGPT, Gemini, Claude** and **Perplexity** is now an essential skill to gain a competitive advantage. Awesome Chapter 19 that clearly introduces the **power of AI API-based app development.** A clear explanation of **Java concurrency.** The **Big O** discussion teaches the costs and consequences of design decisions. Great discussion of creating **custom classes.** Excellent intro to the **Java Platform Module System.** Best intro to **JShell** I've seen. I like the detailed discussion of how **streams and lambdas** work and the thorough explanation of **exceptions.**"

■ **Emily Navarro, Ph.D., Continuing Lecturer, Department of Informatics, University of California, Irvine**

"The book's **"objects-natural" approach** introduces substantial pre-existing Java classes and object interaction before diving into custom class creation in later chapters. I like the **objects-natural** case study with the **date/time** example—a good forced dive into the **Java API** documentation. I like that **inheritance, interfaces,** and **composition** are covered in the same chapter, as it is a logical progression that tells the story of what is the best approach for an OO Java program. The **Building Java Generative AI API-Based Applications** chapter taught me things I did not know. I appreciate

how **AI tools were used to help write the chapter** and the text was transparent about it. The **94 end-of-chapter genAI exercises** are exciting, creative, and fun. I really enjoyed the Files chapter—it brought in a lot of the latest technologies, such as **JSON, CSV, networking APIs,** and **cryptography.** Great job introducing and motivating **generic class and method concepts** and bringing in CS theory to support the practical application of Java's **generic collection data structures.**"

■ **Jeanne Boyarsky, CodeRanch, Java Champion**

"Uses the latest Java features to **write code in the most modern ways.** GenAIs can read diagrams! [Prompt genAIs with class Account's UML class diagram, asking them to generate the corresponding Java class declaration.] **I like the genAI "why questions,"** such as, "prompt genAIs for insights as to why Java constructors cannot return values." I didn't know **GenAI** would tell me to fix the errors even if I didn't ask or could explain how the Java method-call stack works using text-only, step-by-step diagrams made of characters! Cool! Great list of **NLP applications!** The **objects natural NLP, regex and cryptography security examples are awesome!** They let the reader use String knowledge while also learning real-world principles. Includes a substantial treatment of **JavaFX.** The walkthrough of how to set up a JavaFX application in **Scene Builder** was excellent. Great **JavaFX layouts** table. Nice to see **accessibility** in the **JavaFX** chapters. A good intro to **modules**—covers the basics (and more) while remaining approachable. I like that you included the structure of a **modularized application.** The **VideoPlayer module-dependency graph is a great real example.** The progression of running the services example with no providers, then one, then two is a fun way to show the ServiceLoader really picks providers at runtime! I like the number and variety of **JShell** examples and exercises. I like how **lambdas and method references** are presented. Focuses on the most common **generics** use cases and makes them easy to understand. An **objects-natural** case study on **BigDecimal**—interesting discussion of how half-even banker's rounding works for processing **monetary amounts.** I like the **concurrency** examples—the explanations of **virtual threads, structured concurrency** and **scoped values** were good."

Code Snippets

Many titles include programming code or configuration examples. To optimize the presentation of these elements, view the eBook in single-column, landscape mode and adjust the font size to the smallest setting. In addition to presenting code and configurations in the reflowable text format, we have included images of the code that mimic the presentation found in the print book; therefore, where the reflowable format may compromise the presentation of the code listing, you will see a "Click here to view code image" link. Click the link to view the print-fidelity code image. To return to the previous page viewed, click the Back button on your device or app.

```
comments appear like this
keywords appear like this
constants and literal values appear like this
all other code appears in this thin black font
```

```
https://learning.oreilly.com/course/java-fundamentals-with/
   9780135353134/
```

```
https://www.oracle.com/java/technologies/javase/seccodeguide.html
```

```
https://commons.wikimedia.org/wiki/
   Category:SVG_English_pattern_playing_cards
```

```
https://docs.oracle.com/en/java/javase/index.html
```

```
https://docs.oracle.com/en/java/javase/23/docs/api/index.html
```

```
https://docs.oracle.com/javase/specs/
```

```
https://openjdk.org/projects/jdk/
```

```
https://pearson.com/en-us/replocator
```

```
https://amazon.com/
https://barnesandnoble.com/
https://informit.com/
https://vitalsource.com/
https://redshelf.com/
```

```
https://learning.oreilly.com/register
```

```
System.out.println("Welcome to Java!");
```

https://github.com/pdeitel/JavaForProgrammers5e

https://adoptium.net/temurin/releases/

https://adoptium.net/temurin/releases/

OpenJDK23U-jdk_x64_windows_hotspot_23.0.2_7.msi

```
openjdk 23.0.2 2025-01-21
OpenJDK Runtime Environment Temurin-23.0.2+7 (build 23.0.2+7)
OpenJDK 64-Bit Server VM Temurin-23.0.2+7 (build 23.0.2+7, mixed mode, sharing)
```

https://adoptium.net/temurin/releases/

OpenJDK23U-jdk_aarch64_mac_hotspot_23.0.2_7.pkg

OpenJDK23U-jdk_x64_mac_hotspot_23.0.2_7.pkg

```
openjdk 23.0.2 2025-01-21
OpenJDK Runtime Environment Temurin-23.0.2+7 (build 23.0.2+7)
OpenJDK 64-Bit Server VM Temurin-23.0.2+7 (build 23.0.2+7, mixed mode, sharing)
```

https://jetbrains.com/toolbox-app/

https://eclipse.org/downloads/packages/installer

https://code.visualstudio.com/docs/setup/windows

https://code.visualstudio.com/docs/setup/mac

https://code.visualstudio.com/docs/java/java-tutorial

https://gluonhq.com/products/javafx/

openjfx-23.0.2_*OS-architecture*_bin-sdk.zip

C:\Users*User*\Downloads\javafx-sdk-23.0.2\lib

```
export PATH_TO_FX="/Users/User/Downloads/javafx-sdk-23.0.2/lib"
```

```
https://gluonhq.com/labs/scene-builder/
```

What are your capabilities?
What are the differences between your free and paid tiers?

https://blogs.oracle.com/javamagazine/post/the-top-25-greatest-java-apps-ever-written

https://www.jcp.org/en/jsr/all

https://www.jcp.org/en/home/index

https://github.com/topics/java

```
pauldeitel@MacBook-Pro ch01 % javac GuessTheNumberGame.java
pauldeitel@MacBook-Pro ch01 %
```

```
(base) pauldeitel@MacBook-Pro ch01 % java GuessTheNumberGame
I have a number between 1 and 1000.
Can you guess my number?
Please enter your first guess.
?
```

```
(base) pauldeitel@MacBook-Pro ch01 % java GuessTheNumberGame
I have a number between 1 and 1000.
Can you guess my number?
Please enter your first guess.
? 500
Too low. Try again.
?
```

```
(base) pauldeitel@MacBook-Pro ch01 % java GuessTheNumberGame
I have a number between 1 and 1000.
Can you guess my number?
Please enter your first guess.
? 500
Too low. Try again.
? 750
Too high. Try again.
?
```

```
? 625
Too low. Try again.
? 687
Too high. Try again.
? 656
Too high. Try again.
? 640
Too low. Try again.
? 648
Too high. Try again.
? 644
Too high. Try again.
? 642

Excellent! You guessed the number!
Would you like to play again (y or n)? n
```

https://platform.openai.com/docs/guides/prompt-engineering

https://platform.openai.com/docs/examples

```
1  // Fig. 2.1: Welcome1.java
2  // Text-printing program.
3
4  public class Welcome1 {
5     // main method begins program execution
6     public static void main(String[] args) {
7        System.out.println("Welcome to Java Programming!");
8     } // end method main
9  } // end class Welcome1
```

```
Welcome to Java Programming!
```

```
/* This is a traditional comment. It
   can be split over multiple lines */
```

```
cd %userprofile%\Documents\examples\ch02\fig02_01
```

```
cd ~/Documents/examples/ch02/fig02_01
```

```
1  // Fig. 2.2: Welcome2.java
2  // Printing a line of text with multiple statements.
3
4  public class Welcome2 {
5     // main method begins program execution
6     public static void main(String[] args) {
7        System.out.print("Welcome to ");
8        System.out.println("Java Programming!");
9     } // end method main
10 } // end class Welcome2
```

```
Welcome to Java Programming!
```

```
1  // Fig. 2.3: Welcome3.java
2  // Printing multiple lines of text with a single statement.
3
4  public class Welcome3 {
5     // main method begins program execution
6     public static void main(String[] args) {
7        System.out.println("Welcome\nto\nJava\nProgramming!");
8     } // end method main
9  } // end class Welcome3
```

```
Welcome
to
Java
Programming!
```

```
System.out.println("\"in quotes\"");
```

```java
// Fig. 2.4: Welcome4.java
// Displaying multiple lines with method System.out.printf.

public class Welcome4 {
   // main method begins program execution
   public static void main(String[] args) {
      System.out.printf("%s%n%s%n", "Welcome to", "Java Programming!");
   } // end method main
} // end class Welcome4
```

```
Welcome to
Java Programming!
```

```java
System.out.printf("#%n##%n###%n####%n#####%n");
```

```java
// Fig. 2.5: Addition.java
// Addition program that inputs two numbers then displays their sum.
import java.util.Scanner; // program uses class Scanner

public class Addition {
   // main method begins program execution
   public static void main(String[] args) {
      // create a Scanner to obtain input from the user
      Scanner input = new Scanner(System.in);

      System.out.print("Enter first integer: "); // prompt
      int number1 = input.nextInt(); // read first number from user

      System.out.print("Enter second integer: "); // prompt
      int number2 = input.nextInt(); // read second number from user

      int sum = number1 + number2; // add numbers, then store total in sum

      System.out.printf("Sum is %d%n", sum); // display sum
   } // end method main
} // end class Addition
```

```
Enter first integer: 45
Enter second integer: 72
Sum is 117
```

```java
System.out.printf("Sum is %d%n", (number1 + number2));
```

https://docs.oracle.com/en/java/javase/23/docs/api/index.html

https://docs.oracle.com/en/java/javase/23/docs/api/help-doc.html#overview

https://www.oracle.com/java/technologies/javase-jdk23-doc-downloads.html

```java
int number1; // declare the int variable number1
number1 = input.nextInt(); // assign the user's input to number1
```

https://docs.oracle.com/javase/tutorial/java/nutsandbolts/operators.html

```java
int x = 2 % 2 + 2 * 2 - 2 / 2;
```

```java
// Fig. 2.6: Comparison.java
// Compare integers using if statements, relational operators
// and equality operators.
import java.util.Scanner; // program uses class Scanner

public class Comparison {
   // main method begins program execution
   public static void main(String[] args) {
      // create Scanner to obtain input from the user
      Scanner input = new Scanner(System.in);

      System.out.print("Enter first integer: "); // prompt
      int number1 = input.nextInt(); // read first number from user

      System.out.print("Enter second integer: "); // prompt
      int number2 = input.nextInt(); // read second number from user

      if (number1 == number2) {
         System.out.printf("%d == %d%n", number1, number2);
      }

      if (number1 != number2) {
         System.out.printf("%d != %d%n", number1, number2);
      }

      if (number1 < number2) {
         System.out.printf("%d < %d%n", number1, number2);
      }

      if (number1 > number2) {
         System.out.printf("%d > %d%n", number1, number2);
      }

      if (number1 <= number2) {
         System.out.printf("%d <= %d%n", number1, number2);
      }

      if (number1 >= number2) {
         System.out.printf("%d >= %d%n", number1, number2);
      }
   } // end method main
} // end class Comparison
```

```
Enter first integer: 777
Enter second integer: 777
777 == 777
777 <= 777
777 >= 777
```

```
Enter first integer: 1000
Enter second integer: 2000
1000 != 2000
1000 < 2000
1000 <= 2000
```

```
Enter first integer: 2000
Enter second integer: 1000
2000 != 1000
2000 > 1000
2000 >= 1000
```

```java
if (c < 7); {
   System.out.println("c is less than 7");
}

if (c => 7) {
   System.out.println("c is greater than or equal to 7");
}
```

```java
// Fig. 2.7: StringTest.java
// String class test program.
public class StringTest {
   public static void main(String[] args) {
      String s1 = "happy";
      String s2 = " birthday";
      String s3 = ""; // creates an empty string

      // display the Strings and show their lengths
      System.out.printf("s1: \"%s\"; length: %d%n", s1, s1.length());
      System.out.printf("s2: \"%s\"; length: %d%n", s2, s2.length());
      System.out.printf("s3: \"%s\"; length: %d%n%n", s3, s3.length());

      // compare Strings
      System.out.println("Results of comparing Strings:");
      System.out.printf("s1.equals(\"happy\"): %b%n", s1.equals("happy"));
      System.out.printf("s2.equals(s1): %b%n%n", s2.equals(s1));

      // test String method isEmpty
      System.out.println("Testing s3.isEmpty():");

      if (s3.isEmpty()) {
         System.out.println("s3 is empty; assigning to s3");
         s3 = s1 + s2; // assign s3 the result of concatenating s1 and s2

         System.out.printf("s3: \"%s\"; length: %d%n%n", s3, s3.length());
      }

      // testing whether Strings start with "ha" or end with "ay"
      System.out.printf("s1.startsWith(\"ha\"): %b%n",
         s1.startsWith("ha"));
      System.out.printf("s2.startsWith(\"ha\"): %b%n",
         s2.startsWith("ha"));
      System.out.printf("s1.endsWith(\"ay\"): %b%n", s1.endsWith("ay"));
      System.out.printf("s2.endsWith(\"ay\"): %b%n", s2.endsWith("ay"));
   } // end method main
} // end class StringTest
```

```
s1: "happy"; length: 5
s2: " birthday"; length: 9
s3: ""; length: 0

Results of comparing Strings:
s1.equals("happy"): true
s2.equals(s1): false

Testing s3.isEmpty():
s3 is empty; assigning to s3
s3: "happy birthday"; length: 14

s1.startsWith("ha"): true
s2.startsWith("ha"): false
s1.endsWith("ay"): false
s2.endsWith("ay"): true
```

```
if (grade >= 60) {
   System.out.println("Passed");
}
```

```
if (grade >= 60) {
   System.out.println("Passed");
}
else {
   System.out.println("Failed");
}
```

```
if (count > 10) {
   System.out.println("count is greater than 10");
}
else
   System.out.println("count is less than or equal to 10");
}
```

```
if (grade >= 90) {
   System.out.println("A");
}
else {
   if (grade >= 80) {
      System.out.println("B");
   }
   else {
      if (grade >= 70) {
         System.out.println("C");
      }
      else {
         if (grade >= 60) {
            System.out.println("D");
         }
         else {
            System.out.println("F");
         }
      }
   }
}
```

```
if (grade >= 60) {
   System.out.println("Passed");
}
else {
   System.out.println("Failed");
   System.out.println("You must take this course again");
}
```

```
Failed
You must take this course again
```

```
System.out.println("You must take this course again");
```

```
if (grade < 60)
   System.out.println("Failed");
else
   System.out.println("Passed");
   System.out.println("Great job!");
```

```
if (grade < 60) {
   System.out.println("Failed");
}
else {
   System.out.println("Passed");
   System.out.println("Great job!");
}
```

```
System.out.println(grade >= 60 ? "Passed" : "Failed");
```

```
System.out.printf("%d is %s%n",
   number, number % 2 == 0 ? "even" : "odd");
```

```
7 is odd
```

```java
// Power.java
// Calculate the first power of 2 greater than 1,000,000.
public class Power {
   public static void main(String[] args) {
      int product = 2;
      int n = 1;

      while (product <= 1_000_000) {
         product = 2 * product;
         n = n + 1;
      }

      System.out.printf("%s %d%n%s %d%n",
         "First power of 2 greater than 1,000,000 is", product,
         "which is 2 to the power of", n);
   }
}
```

```
First power of 2 greater than 1,000,000 is 1048576
which is 2 to the power of 20
```

```java
// Fig. 3.1: ClassAverage.java
// Solving the class-average problem using counter-controlled iteration.
import java.util.Scanner; // program uses class Scanner

public class ClassAverage {
   public static void main(String[] args) {
      // create Scanner to obtain user input
      Scanner input = new Scanner(System.in);

      // initialization phase
      int total = 0;  // initialize sum of grades entered by the user
      int gradeCounter = 1; // initialize grade # to be entered next

      // processing phase uses counter-controlled iteration
      while (gradeCounter <= 10) { // loop 10 times
         System.out.print("Enter grade: "); // prompt for next grade
         int grade = input.nextInt(); // input the grade
         total = total + grade; // add grade to total
         gradeCounter = gradeCounter + 1; // increment counter by 1
      }

      // termination phase
      int average = total / 10; // integer division yields integer result

      // display total and average of grades
      System.out.printf("%nTotal of all 10 grades is %d%n", total);
      System.out.printf("Class average is %d%n", average);
   }
}
```

```
Enter grade: 67
Enter grade: 78
Enter grade: 89
Enter grade: 67
Enter grade: 87
Enter grade: 98
Enter grade: 93
Enter grade: 85
Enter grade: 82
Enter grade: 100

Total of all 10 grades is 846
Class average is 84
```

```java
total = total + grade; // add grade to total
```

```java
int sum = number1 + number2; // add numbers, then store total in sum
```

https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/lang/Integer.html

```
int grade = input.nextInt(); // input next grade


int count = 1;

while (count < 4) {
    count = count + 1;
    System.out.printf("count is: %d%n", count);
}
```

```
int count = 0;

while (count < 4) {
   count = count + 1;
   System.out.printf("count is: %d%n", count);
}
```

```
count is: 1
count is: 2
count is: 3
count is: 4
```

```
1  // Fig. 3.2: ClassAverage.java
2  // Solving the class-average problem using sentinel-controlled iteration.
3  import java.util.Scanner; // program uses class Scanner
4
5  public class ClassAverage {
6     public static void main(String[] args) {
7        // create Scanner to obtain user input
8        Scanner input = new Scanner(System.in);
9
10       // initialization phase
11       int total = 0; // initialize grade total
12       int gradeCounter = 0; // initialize # of grades entered so far
13
14       // processing phase
15       // prompt for input and read grade from user
16       System.out.print("Enter grade or -1 to quit: ");
17       int grade = input.nextInt();
18
19       // loop until sentinel value read from user
20       while (grade != -1) {
21          total = total + grade; // add grade to total
22          gradeCounter = gradeCounter + 1; // increment counter
23
24          // prompt for input and read next grade from user
25          System.out.print("Enter grade or -1 to quit: ");
```

```
            grade = input.nextInt();
         }

         // termination phase
         // if user entered at least one grade...
         if (gradeCounter != 0) {
            // use a number with decimal point to calculate the average
            double average = (double) total / gradeCounter;

            // display total and average (with two digits of precision)
            System.out.printf("%nTotal of the %d grades entered is %d%n",
               gradeCounter, total);
            System.out.printf("Class average is %.2f%n", average);
         }
         else { // no grades were entered, so output appropriate message
            System.out.println("No grades were entered");
         }
      }
   }
```

```
Enter grade or -1 to quit: 97
Enter grade or -1 to quit: 88
Enter grade or -1 to quit: 72
Enter grade or -1 to quit: -1

Total of the 3 grades entered is 257
Class average is 85.67
```

```
double average = total / gradeCounter;
```

```
// AverageTemperature.java
// Calculate the average of several temperatures
import java.util.Scanner;

public class AverageTemperature {
   public static void main(String[] args) {
      Scanner input = new Scanner(System.in);

      int total = 0;
      int counter = 0;

      System.out.print("Enter a temperature or 999 to quit: ");
      int temperature = input.nextInt();

      while (temperature != 999) {
         total = total + temperature;
         counter = counter + 1;

         System.out.print("Enter a temperature or 999 to quit: ");
         temperature = input.nextInt();
      }

      if (counter != 0) {
         double average = (double) total / counter;
         System.out.printf("Average temperature is: %.2f%n", average);
      }
      else {
         System.out.println("No temperatures were entered.");
      }
   }
}
```

```
Enter a temperature or 999 to quit: 12
Enter a temperature or 999 to quit: -22
Enter a temperature or 999 to quit: 17
Enter a temperature or 999 to quit: 999
Average temperature is: 2.33
```

```java
// Fig. 3.3: Analysis.java
// Analysis of examination results using nested control statements.
import java.util.Scanner; // class uses class Scanner

public class Analysis {
   public static void main(String[] args) {
      // create Scanner to obtain user input
      Scanner input = new Scanner(System.in);

      // initializing variables in declarations
      int passes = 0;
      int failures = 0;
      int studentCounter = 1;

      // process 10 students using counter-controlled iteration
      while (studentCounter <= 10) {
         // prompt user for input and obtain value from user
         System.out.print("Enter result (1 = pass, 2 = fail): ");
         int result = input.nextInt();

         // if...else is nested in the while statement
         if (result == 1) {
            passes = passes + 1;
         }
         else {
            failures = failures + 1;
         }

         // increment studentCounter so loop eventually terminates
         studentCounter = studentCounter + 1;


      }

      // termination phase; prepare and display results
      System.out.printf("Passed: %d%nFailed: %d%n", passes, failures);

      // determine whether more than 8 students passed
      if (passes > 8) {
         System.out.println("Bonus to instructor!");
      }
   }
}
```

```
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 2
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Passed: 9
Failed: 1
Bonus to instructor!
```

```
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 2
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 2
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 2
Enter result (1 = pass, 2 = fail): 2
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Passed: 6
Failed: 4
```

```java
// OddOrEven.java
// Determine whether each integer is odd or even.
import java.util.Scanner;

public class OddOrEven {
   public static void main(String[] args) {
      Scanner input = new Scanner(System.in);
      int counter = 1;

      while (counter <= 3) {
         System.out.print("Enter an integer: ");
         int number = input.nextInt();

         if (number % 2 == 0) {
            System.out.printf("%d is even%n", number);
         }
         else {
            System.out.printf("%d is odd%n", number);
         }

         counter = counter + 1;
      }
   }
}
```

```
Enter an integer: 2
2 is even
Enter an integer: 3
3 is odd
Enter an integer: 5
5 is odd
```

```java
c += 3; // adds 3 to c more concisely
```

```java
// Fig. 3.4: Increment.java
// Prefix increment and postfix increment operators.

public class Increment {
   public static void main(String[] args) {
      // demonstrate postfix increment operator
      int c = 5;
      System.out.printf("c before postincrement: %d%n", c); // prints 5
      System.out.printf("    postincrementing c: %d%n", c++); // prints 5
      System.out.printf(" c after postincrement: %d%n", c); // prints 6

      System.out.println(); // skip a line

      // demonstrate prefix increment operator
      c = 5;
      System.out.printf(" c before preincrement: %d%n", c); // prints 5
      System.out.printf("     preincrementing c: %d%n", ++c); // prints 6
      System.out.printf("  c after preincrement: %d%n", c); // prints 6
   }
}
```

```
c before postincrement: 5
    postincrementing c: 5
 c after postincrement: 6

 c before preincrement: 5
     preincrementing c: 6
  c after preincrement: 6
```

```java
passes = passes + 1;
failures = failures + 1;
studentCounter = studentCounter + 1;
```

Mystery.java

```java
// Mystery.java
public class Mystery {
   public static void main(String[] args) {
      int x = 1;
      int total = 0;

      while (x <= 10) {
         int y = x * x;
         System.out.println(y);
         total += y;
         ++x;
      }

      System.out.printf("Total is %d%n", total);
   }
}
```

https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/math/BigInteger.html

```java
// Fig. 3.5: UsingBigInteger.java
// Creating and manipulating super-sized integers
// with objects of the Java API's BigInteger class.
import java.math.BigInteger;

public class UsingBigInteger {
   public static void main(String[] args) {
      // display the largest value a long can store
      System.out.printf("LARGEST LONG VALUE: %d%n%n", Long.MAX_VALUE);

      // can create BigIntegers from Strings or integers
      BigInteger value1 =
         new BigInteger("10000000000000000000000000000"); // 30 digits
      BigInteger value2 = BigInteger.valueOf(9_223_372_036_854_775_807L);
      BigInteger value3 = BigInteger.valueOf(17);

      System.out.println("INITIAL VALUES");
      System.out.printf("BigInteger value1: %s%n", value1);
      System.out.printf("BigInteger value2: %s%n", value2);
      System.out.printf("BigInteger value3: %s%n", value3);

      System.out.println("\nADD, SUBTRACT AND MULTIPLY BIGINTEGERS");
      System.out.printf("     value1.add(value2): %s%n",
         value1.add(value2));
      System.out.printf("value1.subtract(value2): %s%n",
         value1.subtract(value2));
      System.out.printf("value1.multiply(value2): %s%n",
         value1.multiply(value2));
      System.out.printf("value1.multiply(value3): %s%n",
         value1.multiply(value3));
   }
}
```

```
LARGEST TYPE LONG VALUE: 9223372036854775807

INITIAL VALUES
BigInteger value1: 10000000000000000000000000000
BigInteger value2: 9223372036854775807
BigInteger value3: 17

ADD, SUBTRACT AND MULTIPLY BIGINTEGERS
     value1.add(value2): 10000000000922337203685477580 7
value1.subtract(value2): 9999999990776627963145224193
value1.multiply(value2): 92233720368547758070000000000000000000000000000
value1.multiply(value3): 17000000000000000000000000000
```

```java
BigInteger value = new BigInteger("100000000000000");
System.out.printf("      value is: %s%nvalue cubed is: %s%n",
   value, value.multiply(value.multiply(value)));
```

```
      value is: 100000000000000
value cubed is: 1000000000000000000000000000000000000000000000
```

```java
// Fig. 4.1: WhileCounter.java
// Counter-controlled iteration with the while iteration statement.

public class WhileCounter {
   public static void main(String[] args) {
      int counter = 1; // declare and initialize control variable

      while (counter <= 10) { // loop-continuation condition
         System.out.printf("%d  ", counter);
         ++counter; // increment control variable
      }

      System.out.println();
   }
}
```

```
1  2  3  4  5  6  7  8  9  10
```

```java
// Fig. 4.2: ForCounter.java
// Counter-controlled iteration with the for iteration statement.

public class ForCounter {
   public static void main(String[] args) {
      // for statement header includes initialization, loop-continuation
      // condition and increment separated by required semicolons
      for (int counter = 1; counter <= 10; ++counter) {
         System.out.printf("%d  ", counter);
      }

      System.out.println();
   }
}
```

```
1  2  3  4  5  6  7  8  9  10
```

```
for (initialization; loopContinuationCondition; increment) {
   statement(s)
}
```

```java
for (int i = 1; i <= 100; ++i)

for (int i = 100; i >= 1; --i)

for (int i = 7; i <= 77; i += 7)

for (int i = 20; i >= 2; i -= 2)

for (int i = 2; i <= 20; i += 3)

for (int i = 99; i >= 0; i -= 11)
```

```
for (int counter = 1; counter != 10; counter += 2)
```

```
for (int i = 19; i >= 1; i += 2) {
    System.out.println(i);
}
```

```
1  // Fig. 4.3: Sum.java
2  // Summing integers with the for statement.
3
4  public class Sum {
5     public static void main(String[] args) {
6        int total = 0;
7
8        // total even integers from 2 through 20
9        for (int number = 2; number <= 20; number += 2) {
10          total += number;
11       }
12
13       System.out.printf("Sum is %d%n", total);
14    }
15 }
```

```
Sum is 110
```

```
int sum = 0;

for (int count = 1; count <= 99; count += 2) {
   sum += count;
}

System.out.printf("Sum is: %d%n", sum);
```

```
Sum is: 2500
```

```
1  // Fig. 4.4: Interest.java
2  // Compound-interest calculations with for.
3
4  public class Interest {
5     public static void main(String[] args) {
6        double principal = 1000.00; // initial amount before interest
7        double rate = 0.05; // interest rate
8
9        // display headers
10       System.out.printf("%s%20s%n", "Year", "Amount on deposit");
11
12       // calculate amount on deposit for each of ten years
13       for (int year = 1; year <= 10; ++year) {
14          // calculate new amount on deposit for specified year
15          double amount = principal * Math.pow(1.0 + rate, year);
16
17          // display the year and the amount
18          System.out.printf("%4d%,20.2f%n", year, amount);
19       }
20    }
21 }
```

```
Year   Amount on deposit
   1           1,050.00
   2           1,102.50
   3           1,157.63
   4           1,215.51
   5           1,276.28
   6           1,340.10
   7           1,407.10
   8           1,477.46
   9           1,551.33
  10           1,628.89
```

```java
// Fig. 4.5: DoWhileTest.java
// do...while iteration statement.

public class DoWhileTest {
   public static void main(String[] args) {
      int counter = 1;

      do {
         System.out.printf("%d  ", counter);
         ++counter;
      } while (counter <= 10);

      System.out.println();
   }
}
```

```
1  2  3  4  5  6  7  8  9  10
```

```java
int counter = 2;

do {
   System.out.println(counter);
   counter += 2;
} While (counter < 10);
```

```java
// Fig. 4.6: LetterGrades.java
// LetterGrades class uses the switch statement to count letter grades.
import java.util.Scanner;

public class LetterGrades {
   public static void main(String[] args) {
      int total = 0; // sum of grades
      int gradeCounter = 0; // number of grades entered
      int aCount = 0; // count of A grades
      int bCount = 0; // count of B grades
      int cCount = 0; // count of C grades
      int dCount = 0; // count of D grades
      int fCount = 0; // count of F grades

      Scanner input = new Scanner(System.in);

      System.out.println("""
         Enter the integer grades in the range 0-100.
         Enter the end-of-file indicator to terminate input:
            On macOS/Linux enter <Ctrl> d then press Enter
            On Windows enter <Ctrl> z then press Enter""");

      // loop until user enters the end-of-file indicator
      while (input.hasNext()) {
         int grade = input.nextInt(); // read grade
         total += grade; // add grade to total
         ++gradeCounter; // increment number of grades

         //  increment appropriate letter-grade counter
         switch (grade / 10) {
            case 10:  // grade was 90-100, inclusive
            case 9:
               ++aCount;
               break; // exits switch
            case 8: // grade was 80-89
               ++bCount;
               break; // exits switch
            case 7: // grade was 70-79
```

```
                ++cCount;
                break; // exits switch
             case 6: // grade was 60-69
                ++dCount;
                break; // exits switch
             default: // grade was less than 60
                ++fCount;
                break; // optional; exits switch anyway
          }
       }

       // display grade report
       System.out.printf("%nGrade Report:%n");

       // if user entered at least one grade...
       if (gradeCounter != 0) {
          // calculate average of all grades entered
          double average = (double) total / gradeCounter;

          // output summary of results
          System.out.printf("Total of the %d grades entered is %d%n",
             gradeCounter, total);
          System.out.printf("Class average is %.2f%n", average);
          System.out.printf("%n%s%n%s%d%n%s%d%n%s%d%n%s%d%n%s%d%n",
             "Number of students who received each grade:",
             "A: ", aCount,  // display number of A grades
             "B: ", bCount,  // display number of B grades
             "C: ", cCount,  // display number of C grades
             "D: ", dCount,  // display number of D grades
             "F: ", fCount); // display number of F grades
       }
       else { // no grades were entered, so output appropriate message
          System.out.println("No grades were entered");
       }
    }
 }
```

```
Enter the integer grades in the range 0-100.
Enter the end-of-file indicator to terminate input:
   On macOS/Linux enter <Ctrl> d then press Enter
   On Windows enter <Ctrl> z then press Enter
99
92
45
57
63
71
76
85
90
100
^Z

Grade Report:
Total of the 10 grades entered is 778
Class average is 77.80

Number of students who received each grade:
A: 4
B: 1
C: 2
D: 1
F: 2
```

https://en.wikipedia.org/wiki/List_of_Unicode_characters

```
switch (number) {
   case 1:
      System.out.println("The number is 1\n");
   case 2:
      System.out.println("The number is 2\n");
      break;
   default:
      System.out.println("The number is not 1 or 2\n");
      break;
}
```

```
1  // Fig. 4.7: BreakTest.java
2  // break statement exiting a for statement.
3  public class BreakTest {
4     public static void main(String[] args) {
5        int count; // control variable also used after loop terminates
6
7        for (count = 1; count <= 10; ++count) { // loop 10 times
8           if (count == 5) {
9              break; // terminates loop if count is 5
10          }
11
12          System.out.printf("%d ", count);
13       }
14
15       System.out.printf("%nBroke out of loop at count = %d%n", count);
16    }
17 }
```

```
1 2 3 4
Broke out of loop at count = 5
```

```
1  // Fig. 4.8: ContinueTest.java
2  // continue statement terminating an iteration of a for statement.
3  public class ContinueTest {
4     public static void main(String[] args) {
5        for (int count = 1; count <= 10; ++count) { // loop 10 times
6           if (count == 5) {
7              continue; // skip remaining code in loop body if count is 5
8           }
9
10          System.out.printf("%d ", count);
11       }
12
13       System.out.printf("%nUsed continue to skip printing 5%n");
14    }
15 }
```

```
1 2 3 4 6 7 8 9 10
Used continue to skip printing 5
```

```
for (int count = 1; count <= 10; ++count) { // loop 10 times
   if (count != 5) {
      System.out.printf("%d ", count);
   }
}
```

```
if (employeeType == PILOT && age >= 63) {
   ++veteranPilots;
}
```

```
if ((employeeType == PILOT) && (age >= 63)) {
   ++veteranPilots;
}
```

```java
if ((semesterAverage >= 90) || (finalExam >= 90)) {
   System.out.println ("Student grade is A");
}
```

```java
(semesterAverage >= 90) || (finalExam >= 90)
```

```java
(employeeType == PILOT) && (age >= 63)
```

```java
(birthday == true) | (++age >= 65)
```

```java
if (!(grade == sentinelValue)) {
   System.out.printf("The next grade is %d%n", grade);
}
```

```java
if (grade != sentinelValue) {
   System.out.printf("The next grade is %d%n", grade);
}
```

```java
// Fig. 4.9: LogicalOperators.java
// Logical operators.

public class LogicalOperators {
   public static void main(String[] args) {
      // create truth table for && (conditional AND) operator
      System.out.printf("%s%n%s: %b%n%s: %b%n%s: %b%n%n",
         "Conditional AND (&&)", "false && false", (false && false),
         "false && true", (false && true),
         "true && false", (true && false),
         "true && true", (true && true));

      // create truth table for || (conditional OR) operator
      System.out.printf("%s%n%s: %b%n%s: %b%n%s: %b%n%n",
         "Conditional OR (||)", "false || false", (false || false),
         "false || true", (false || true),
         "true || false", (true || false),
         "true || true", (true || true));

      // create truth table for & (boolean logical AND) operator
      System.out.printf("%s%n%s: %b%n%s: %b%n%s: %b%n%n",
         "Boolean logical AND (&)", "false & false", (false & false),
         "false & true", (false & true),
         "true & false", (true & false),
         "true & true", (true & true));

      // create truth table for | (boolean logical inclusive OR) operator
      System.out.printf("%s%n%s: %b%n%s: %b%n%s: %b%n%n",
         "Boolean logical inclusive OR (|)",
         "false | false", (false | false),
         "false | true", (false | true),
         "true | false", (true | false),
         "true | true", (true | true));

      // create truth table for ^ (boolean logical exclusive OR) operator
      System.out.printf("%s%n%s: %b%n%s: %b%n%s: %b%n%n",
```

```
            "Boolean logical exclusive OR (^)",
            "false ^ false", (false ^ false),
            "false ^ true", (false ^ true),
            "true ^ false", (true ^ false),
            "true ^ true", (true ^ true));

        // create truth table for ! (logical negation) operator
        System.out.printf("%s%n%s: %b%n%s: %b%n", "Logical NOT (!)",
            "!false", (!false), "!true", (!true));
    }
}
```

```
Conditional AND (&&)
false && false: false
false && true: false
true && false: false
true && true: true

Conditional OR (||)
false || false: false
false || true: true
true || false: true
true || true: true

Boolean logical AND (&)
false & false: false
false & true: false
true & false: false
true & true: true

Boolean logical inclusive OR (|)
false | false: false
false | true: true
true | false: true
true | true: true

Boolean logical exclusive OR (^)
false ^ false: false
false ^ true: true
true ^ false: true
true ^ true: false

Logical NOT (!)
!false: true
!true: false
```

```
BigDecimal monetaryAmount = new BigDecimal(123.02);
```

123.01999999999999602096067974343895912170410156 25

```
BigDecimal monetaryAmount = new BigDecimal("123.02");
```

```java
// Fig. 4.11: BigDecimalInterest.java
// Compound-interest example with BigDecimal objects.
import java.math.BigDecimal;
import java.math.RoundingMode;

public class BigDecimalInterest {
   public static void main(String[] args) {
      // create BigDecimal objects from strings
      BigDecimal principal = new BigDecimal("1000.00");
      BigDecimal rate = new BigDecimal("0.05");

      System.out.printf("Initial principal: %7s%n", principal);
      System.out.printf("    Interest rate: %7s%n%n", rate);

      // display headers
      System.out.printf("%s%17s   %s%n",
         "Year", "Rounded Amount", "Precise Stored Amount");

      // calculate amount on deposit for each of 10 years
      for (int year = 1; year <= 10; ++year) {
         BigDecimal amount = principal.multiply(
            BigDecimal.ONE.add(rate).pow(year));
         System.out.printf("%4d%17s   %s%n", year,
            amount.setScale(2, RoundingMode.HALF_EVEN), amount);
      }
   }
}
```

```
Initial principal: 1000.00
    Interest rate:    0.05

Year   Rounded Amount   Precise Stored Amount
   1          1050.00   1050.0000
   2          1102.50   1102.500000
   3          1157.62   1157.62500000
   4          1215.51   1215.5062500000
   5          1276.28   1276.281562500000
   6          1340.10   1340.09564062500000
   7          1407.10   1407.1004226562500000
   8          1477.46   1477.455443789062500000
   9          1551.33   1551.32821597851562500000
  10          1628.89   1628.894626777441062500000
```

```
BigDecimal amount = principal.multiply(
   BigDecimal.ONE.add(rate).pow(year));
```

```
amount.setScale(2, RoundingMode.HALF_EVEN)
```

```java
// Fig. 5.1: MaximumFinder.java
// Programmer-declared method maximum with three double parameters.
import java.util.Scanner;

public class MaximumFinder {
   public static void main(String[] args) {
      // create Scanner for input from command window
      var input = new Scanner(System.in);

      // prompt for and input three floating-point values
      System.out.print(
         "Enter three floating-point values separated by spaces: ");
      double number1 = input.nextDouble(); // read first double
      double number2 = input.nextDouble(); // read second double
      double number3 = input.nextDouble(); // read third double

      // determine the maximum value
      double result = maximum(number1, number2, number3);

      // display maximum value
      System.out.printf("Maximum is: %.2f%n", result);
   }


   // returns the maximum of its three double parameters
   public static double maximum(double x, double y, double z) {
      double maximumValue = x; // assume x is the largest to start

      // determine whether y is greater than maximumValue
      if (y > maximumValue) {
         maximumValue = y;
      }

      // determine whether z is greater than maximumValue
      if (z > maximumValue) {
         maximumValue = z;
      }

      return maximumValue;
   }
}
```

```
Enter three floating-point values separated by spaces: 9.35 2.74 5.1
Maximum is: 9.35
```

```
Enter three floating-point values separated by spaces: 5.8 12.45 8.32
Maximum is: 12.45
```

```
Enter three floating-point values separated by spaces: 6.46 4.12 10.54
Maximum is: 10.54
```

```java
MaximumFinder.maximum(10, 30, 20)
```

```java
Scanner input = new Scanner(System.in);
```

```java
return Math.max(x, Math.max(y, z));
```

https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/util/random/package-summary.html

```java
RandomGenerator randomNumbers = RandomGenerator.getDefault()
```

https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/util/random/RandomGenerator.html

```
int randomValue = randomNumbers.nextInt();

int randomValue = randomNumbers.nextInt(2);

int randomValue = randomNumbers.nextInt(1, 7);
```

```
1  // Fig. 5.2: RandomIntegers.java
2  // Rolling a six-sided die.
3  import java.util.random.RandomGenerator;
4
5  public class RandomIntegers {
6     public static void main(String[] args) {
7        // randomNumbers object will produce random numbers
8        RandomGenerator randomNumbers = RandomGenerator.getDefault();
9
10       // loop 10 times
11       for (int counter = 1; counter <= 10; ++counter) {
12          // pick random integer from 1 to 6
13          int face = randomNumbers.nextInt(1, 7);
14
15          System.out.printf("%d  ", face); // display generated value
16       }
17
18       System.out.println();
19    }
20 }
```

```
1  1  3  6  4  2  5  5  3  4
```

```
3  6  3  5  3  2  6  6  4  5
```

```
1  // Fig. 5.3: RollDie.java
2  // Roll a six-sided die 60,000,000 times.
3  import java.util.random.RandomGenerator;
4
5  public class RollDie {
6     public static void main(String[] args) {
7        // randomNumbers object will produce random numbers
8        RandomGenerator randomNumbers = RandomGenerator.getDefault();
9
10       int frequency1 = 0; // count of 1s rolled
11       int frequency2 = 0; // count of 2s rolled
12       int frequency3 = 0; // count of 3s rolled
13       int frequency4 = 0; // count of 4s rolled
14       int frequency5 = 0; // count of 5s rolled
15       int frequency6 = 0; // count of 6s rolled
16
17       // tally counts for 60,000,000 rolls of a die
18       for (int roll = 1; roll <= 60_000_000; ++roll) {
19          int face = randomNumbers.nextInt(1, 7); // number from 1 to 6
20
21          // use face value 1-6 to determine which counter to increment
22          switch (face) {
23             case 1 -> ++frequency1; // increment the 1s counter
24             case 2 -> ++frequency2; // increment the 2s counter
25             case 3 -> ++frequency3; // increment the 3s counter
26             case 4 -> ++frequency4; // increment the 4s counter
27             case 5 -> ++frequency5; // increment the 5s counter
28             case 6 -> ++frequency6; // increment the 6s counter
```

```
            }
         }

         System.out.println("Face\tFrequency"); // output headers
         System.out.printf("1\t%d%n2\t%d%n3\t%d%n4\t%d%n5\t%d%n6\t%d%n",
            frequency1, frequency2, frequency3, frequency4,
            frequency5, frequency6);
      }
   }
```

```
Face    Frequency
1       9996511
2       9999764
3       10001371
4       9997276
5       10003795
6       10001283
```

```
Face    Frequency
1       9999894
2       10000834
3       10000705
4       9997478
5       9999072
6       10002017
```

```
int number = 2 + 3 * randomNumbers.nextInt(5);
```

```
int number = shiftingValue +
   differenceBetweenValues * randomNumbers.nextInt(scalingFactor);
```

```
var randomNumbers = new SecureRandom();
```

```java
// Fig. 5.4: Craps.java
// Craps class simulates the dice game craps.
import java.util.random.RandomGenerator;

public class Craps {
   // get the default RandomGenerator for use in method rollDice
   private static final RandomGenerator randomNumbers =
      RandomGenerator.getDefault();

   // enum type with constants that represent the game status
   private enum Status {CONTINUE, WON, LOST};

   // plays one game of craps
   public static void main(String[] args) {
      int myPoint = 0; // point if no win or loss on first roll
      Status gameStatus; // can contain CONTINUE, WON or LOST

      int sumOfDice = rollDice(); // first roll of the dice

      // determine game status and point based on first roll
      switch (sumOfDice) {
         case 7:  // win with 7 on first roll
         case 11: // win with 11 on first roll
            gameStatus = Status.WON;
            break;
         case 2: // lose with 2 on first roll
         case 3: // lose with 3 on first roll
         case 12: // lose with 12 on first roll
            gameStatus = Status.LOST;
            break;
         default: // did not win or lose, so remember point
            gameStatus = Status.CONTINUE; // game is not over
            myPoint = sumOfDice; // remember the point
            System.out.printf("Point is %d%n", myPoint);
            break;
      }

      // while game is not complete
      while (gameStatus == Status.CONTINUE) { // not WON or LOST
         sumOfDice = rollDice(); // roll dice again

         // determine game status
         if (sumOfDice == myPoint) { // win by making point
            gameStatus = Status.WON;
         }
         else {
            if (sumOfDice == 7) { // lose by rolling 7 before point
               gameStatus = Status.LOST;
            }
         }
      }

```

```
53          // display won or lost message
54          if (gameStatus == Status.WON) {
55             System.out.println("Player wins");
56          }
57          else {
58             System.out.println("Player loses");
59          }
60       }
61
62       // roll dice, calculate sum and display results
63       public static int rollDice() {
64          // pick random die values
65          int die1 = randomNumbers.nextInt(1, 7); // first die roll
66          int die2 = randomNumbers.nextInt(1, 7); // second die roll
67
68          int sum = die1 + die2; // sum of die values
69
70          // display results of this roll
71          System.out.printf("Player rolled %d + %d = %d%n", die1, die2, sum);
72
73          return sum;
74       }
75    }
```

```
Player rolled 5 + 6 = 11
Player wins
```

```
Player rolled 1 + 2 = 3
Player loses
```

```
Player rolled 5 + 4 = 9
Point is 9
Player rolled 4 + 2 = 6
Player rolled 3 + 6 = 9
Player wins
```

```
Player rolled 2 + 6 = 8
Point is 8
Player rolled 5 + 1 = 6
Player rolled 2 + 1 = 3
Player rolled 1 + 6 = 7
Player loses
```

```
Status gameStatus = switch (sumOfDice) {
   case 7, 11 -> Status.WON;
   case 2, 3, 12 -> Status.LOST;
   default -> {
      myPoint = sumOfDice; // remember the point
      System.out.printf("Point is %d%n", myPoint);
      yield Status.CONTINUE; // game is not over
   }
}; // required semicolon to terminate the statement
```

```
// determine game status
gameStatus = switch (sumOfDice) {
   case 7 -> Status.LOST;
   default -> sumOfDice == myPoint ?
      Status.WON : Status.CONTINUE;
};
```

```
String result = switch (flip) {
   case 0 -> "heads";
   case 1 -> "tails";
   default -> {
      System.out.printf("%d is invalid%n", flip);
      yield "";
   }
};
```

```
1  // Fig. 5.5: Scope.java
2  // Scope class demonstrates field and local-variable scopes.
3
4  public class Scope {
5     // field that is accessible to all methods of this class
6     private static int x = 1;
7
8     // method main creates and initializes local variable x
9     // and calls methods useLocalVariable and useField
10    public static void main(String[] args) {
11       int x = 5; // method's local variable x shadows field x
12
13       System.out.printf("local x in main is %d%n", x);
14
15       useLocalVariable(); // useLocalVariable has local x
16       useField(); // useField uses class Scope's field x
17       useLocalVariable(); // useLocalVariable reinitializes local x
18       useField(); // class Scope's field x retains its value
19
20       System.out.printf("%nlocal x in main is %d%n", x);
21    }
22
23    // create and initialize local variable x during each call
24    public static void useLocalVariable() {
25       int x = 25; // initialized each time useLocalVariable is called
26
27       System.out.printf(
28          "%nlocal x on entering method useLocalVariable is %d%n", x);
29       ++x; // modifies this method's local variable x
30       System.out.printf(
31          "local x before exiting method useLocalVariable is %d%n", x);
32    }
33
34    // modify class Scope's field x during each call
35    public static void useField() {
36       System.out.printf(
37          "%nfield x on entering method useField is %d%n", x);
38       x *= 10; // modifies class Scope's field x
39       System.out.printf(
40          "field x before exiting method useField is %d%n", x);
41    }
42 }
```

```
local x in main is 5

local x on entering method useLocalVariable is 25
local x before exiting method useLocalVariable is 26

field x on entering method useField is 1
field x before exiting method useField is 10

local x on entering method useLocalVariable is 25
local x before exiting method useLocalVariable is 26

field x on entering method useField is 10
field x before exiting method useField is 100

local x in main is 5
```

```java
// Fig. 5.6: MethodOverload.java
// Overloaded method declarations.

public class MethodOverload {
   // test overloaded square methods
   public static void main(String[] args) {
      System.out.printf("Square of integer 7 is %d%n", square(7));
      System.out.printf("Square of double 7.5 is %f%n", square(7.5));
   }

   // square method with int argument
   public static int square(int intValue) {
      System.out.printf("%nCalled square with int argument: %d%n",
         intValue);
      return intValue * intValue;
   }

   // square method with double argument
   public static double square(double doubleValue) {
      System.out.printf("%nCalled square with double argument: %f%n",
         doubleValue);
      return doubleValue * doubleValue;
   }
}
```

```
Called square with int argument: 7
Square of integer 7 is 49

Called square with double argument: 7.500000
Square of double 7.5 is 56.250000
```

```java
public static double multiply(double x, double y) {
   return x * y;
}

public static int multiply(int x, int y) {
   return x * y;
}
```

```java
System.out.println(Math.sqrt(900.0));

System.out.println(Math.sqrt(c + d * f));
```

```
java ClassName argument1 argument2 ...
```

https://docs.oracle.com/en/java/javase/23/docs/api/java.base/module-summary.html

https://docs.oracle.com/en/java/javase/23/docs/api/help-doc.html

```java
// Fig. 5.7: SquareDemo.java
// Square method used to demonstrate the
// method-call stack and activation records.

public class SquareDemo {
   public static void main(String[] args) {
      int y = 10; // value to square (local variable in main)

      int result = square(y); // calculate y squared and store in result
   }

   // returns the square of an integer
   public static int square(int x) {
      return x * x; // calculate square and return result
   }
}
```

```
System.out.println(Math.sqrt(4));
```

```
Enter your birthday (yyyy-MM-dd): 1969-07-20
Enter your birth time (HH:mm): 22:56

Birth date/time: Jul 20, 1969, 10:56:00 PM
Current date/time: Jun 15, 2024, 12:40:56 PM

You were born on a Sunday
You have been alive for 54 years, 10 months, and 26 days

Total months:  658
Total days:    20,054
Total hours:   481,285
Total minutes: 28,877,144
Total seconds: 1,732,628,696
```

```
1  // Fig. 5.8: AgeCalculator.java
2  // Using the Date/Time API to calculate how long the user has
3  // been alive and determine the day of the week on which they were born.
4  import java.time.Duration;
5  import java.time.LocalDate;
6  import java.time.LocalDateTime;
7  import java.time.LocalTime;
8  import java.time.Period;
9  import java.time.format.DateTimeFormatter;
10 import java.time.format.FormatStyle;
11 import java.time.format.TextStyle;
12 import java.time.temporal.ChronoUnit;
13 import java.util.Locale;
14 import java.util.Scanner;
15
```

```
16 public class AgeCalculator {
17    public static void main(String[] args) {
18       var input = new Scanner(System.in);
19
20       // input user's birthday and convert it to a LocalDate
21       System.out.print("Enter your birthday (yyyy-MM-dd): ");
22       var birthday = LocalDate.parse(input.nextLine());
23
24       // input user's birth time and convert it to a LocalTime
25       System.out.print("Enter your birth time (HH:mm): ");
26       var birthTime = LocalTime.parse(input.nextLine());
27
```

```
Enter your birthday (yyyy-MM-dd): 1969-07-20
Enter your birth time (HH:mm): 22:56
```

```
28       // combine the birthday and birthTime into a LocalDateTime object
29       var birthDateTime = LocalDateTime.of(birthday, birthTime);
30
31       // get the current date and time as a LocalDateTime object
32       var now = LocalDateTime.now();
33
34       // show the dates/times
35       displayDatesAndTimes(birthDateTime, now);
36
```

```
Birth date/time: Jul 20, 1969, 10:56:00 AM
Current date/time: Jun 15, 2024, 11:40:59 AM
```

```
37       // calculate date/time difference in years, months and days
38       var period = Period.between(birthday, now.toLocalDate());
39
40       // calculate date/time difference in hours, minutes and seconds
41       var duration = Duration.between(birthDateTime, now);
42
```

```
43          // display day of the week the user was born
44          System.out.printf("You were born on a %s%n%n",
45             birthday.getDayOfWeek().getDisplayName(
46                TextStyle.FULL, Locale.getDefault()));
47
```

```
You were born on a Sunday
```

```
48          // display age components
49          System.out.printf("""
50             You have been alive for %d years, %d months, and %d days
51
52                 Total months: %,d
53                   Total days: %,d
54                  Total hours: %,d
55                Total minutes: %,d
56                Total seconds: %,d
57             """,
58             period.getYears(), period.getMonths(), period.getDays(),
59             period.toTotalMonths(),
60             birthday.until(now.toLocalDate(), ChronoUnit.DAYS),
61             duration.toHours(), duration.toMinutes(), duration.toSeconds());
62       }
63
```

```
You have been alive for 54 years, 10 months, and 26 days

Total months:  658
Total days:    20,054
Total hours:   481,285
Total minutes: 28,877,144
Total seconds: 1,732,628,696
```

```
64       // display birth date/time and current date/time in user's locale
65       public static void displayDatesAndTimes(
66          LocalDateTime birth, LocalDateTime current) {
67
68          // prepare format to display dates and times for user's locale
69          var formatter =
70             DateTimeFormatter.ofLocalizedDateTime(FormatStyle.MEDIUM);
71
72          // display formatted date and time
73          System.out.printf("%nBirth date/time: %s%n",
74             birth.format(formatter));
75          System.out.printf("Current date/time: %s%n%n",
76             current.format(formatter));
77       }
78    }
```

```
java -Duser.language=es -Duser.country=ES
   -Duser.timezone=Europe/Madrid AgeCalculator
```

```
Enter your birthday (yyyy-MM-dd): 1969-07-21
Enter your birth time (HH:mm): 03:56

Birth date/time: 21 jul 1969, 3:56:00
Current date/time: 15 jun 2024, 17:42:39

You were born on a lunes
You have been alive for 54 years, 10 months, and 25 days

Total months:  658
Total days:    20.053
Total hours:   481.285
Total minutes: 28.877.146
Total seconds: 1.732.628.799
```

https://en.wikipedia.org/wiki/List_of_ISO_639_language_codes

https://en.wikipedia.org/wiki/ISO_3166-1_alpha-2

https://en.wikipedia.org/wiki/List_of_tz_database_time_zones

```
int[] c; // declare the array variable
c = new int[5]; // create the array; assign to array variable
```

```
String[] b = new String[100], x = new String[27];
```

```
String[] b = new String[100]; // create array b
String[] x = new String[27]; // create array x
```

```
1  // Fig. 6.1: InitArray.java
2  // Initializing the elements of an array to default values of zero.
3
4  public class InitArray {
5     public static void main(String[] args) {
6        // declare variable array and initialize it with an array object
7        int[] array = new int[5]; // create the array object
8
9        System.out.printf("%s%8s%n", "Index", "Value"); // column headings
10
11       // output each array element's value
12       for (int counter = 0; counter < array.length; ++counter) {
13          System.out.printf("%5d%8d%n", counter, array[counter]);
14       }
15    }
16 }
```

```
Index   Value
    0       0
    1       0
    2       0
    3       0
    4       0
```

```
int[] values = new int[10];

for (int i = 0; i <= values.length; ++i) {
   System.out.println(values[i]);
}
```

```
int[] n = {10, 20, 30, 40, 50};
```

```
1  // Fig. 6.2: InitArray.java
2  // Initializing the elements of an array with an array initializer.
3
4  public class InitArray {
5     public static void main(String[] args) {
6        // array initializer specifies the initial value for each element
7        int[] array = {32, 27, 64, 18, 95};
8
9        System.out.printf("%s%8s%n", "Index", "Value"); // column headings
10
11       // output each array element's value
12       for (int counter = 0; counter < array.length; ++counter) {
13          System.out.printf("%5d%8d%n", counter, array[counter]);
14       }
15    }
16 }
```

```
Index   Value
    0      32
    1      27
    2      64
    3      18
    4      95
```

```
for (int counter = 0; counter < values.length; ++counter) {
   otherValues[counter] = values[counter];
}
```

```
1  // Fig. 6.3: InitArray.java
2  // Calculating the values to be placed into the elements of an array.
3
4  public class InitArray {
5     public static void main(String[] args) {
6        final int ARRAY_LENGTH = 5; // declare constant
7        int[] array = new int[ARRAY_LENGTH]; // create array
8
9        // calculate value for each array element
10       for (int counter = 0; counter < array.length; ++counter) {
11          array[counter] = 2 + 2 * counter;
12       }
13
14       System.out.printf("%s%8s%n", "Index", "Value"); // column headings
15
16       // output each array element's value
17       for (int counter = 0; counter < array.length; ++counter) {
18          System.out.printf("%5d%8d%n", counter, array[counter]);
19       }
20    }
21 }
```

```
Index   Value
    0       2
    1       4
    2       6
    3       8
    4      10
```

```
1  // Fig. 6.4: SumArray.java
2  // Computing the sum of the elements of an array.
3
4  public class SumArray {
5     public static void main(String[] args) {
6        int[] grades = {87, 68, 94, 100, 83, 78, 85, 91, 76, 87};
7        int total = 0;
8
9        // add each element's value to total
10       for (int counter = 0; counter < grades.length; ++counter) {
11          total += grades[counter];
12       }
13
14       System.out.printf("Total of array elements: %d%n", total);
15    }
16 }
```

```
Total of array elements: 849
```

```java
// Fig. 6.5: BarChart.java
// Bar chart printing program.

public class BarChart {
   public static void main(String[] args) {
      int[] distribution = {0, 0, 0, 0, 0, 0, 1, 2, 4, 2, 1};

      System.out.println("Grade distribution:");

      // for each distribution element, output a bar of the chart
      for (int counter = 0; counter < distribution.length; ++counter) {
         // output bar label ("00-09: ", ..., "90-99: ", "100: ")
         if (counter == 10) {
            System.out.printf("%5d: ", 100);
         }
         else {
            System.out.printf("%02d-%02d: ",
               counter * 10, counter * 10 + 9);
         }

         // print bar of asterisks
         for (int stars = 0; stars < distribution[counter]; ++stars) {
            System.out.print("*");
         }

         System.out.println();
      }
   }
}
```

```
Grade distribution:
00-09:
10-19:
20-29:
30-39:
40-49:
50-59:
60-69: *
70-79: **
80-89: ****
90-99: **
  100: *
```

```java
System.out.println("*".repeat(distribution[counter]));
```

```java
// Fig. 6.6: RollDie.java
// Die-rolling program using arrays instead of switch.
import java.util.random.RandomGenerator;

public class RollDie {
   public static void main(String[] args) {
      var randomNumbers = RandomGenerator.getDefault();
      int[] frequency = new int[7]; // array of frequency counters

      // roll die 60,000,000 times; use die value as frequency index
      for (int roll = 1; roll <= 60_000_000; ++roll) {
         ++frequency[randomNumbers.nextInt(1, 7)];
      }

      System.out.printf("%s%12s%n", "Face", "Frequency");

      // output each array element's value
      for (int face = 1; face < frequency.length; ++face) {
         System.out.printf("%4d%12d%n", face, frequency[face]);
      }
   }
}
```

```
Face   Frequency
   1    10001418
   2     9999022
   3     9996511
   4    10001187
   5    10002132
   6     9999730
```

```java
// CoinFlipper.java
// Summarizing coin flips.
import java.util.random.RandomGenerator;

public class CoinFlipper {
   public static void main(String[] args) {
      RandomGenerator randomNumbers = RandomGenerator.getDefault();

      // constants representing heads and tails
      final int HEADS = 0;
      final int TAILS = 1;

      // frequency[0] for heads, frequencies[1] for tails
      int[] frequency = new int[2];

      // simulate 100 coin flips
      for (int counter = 0; counter < 100; ++counter) {
         ++frequency[randomNumbers.nextInt(2)];
      }

      System.out.printf("Number of heads: %d%n", frequency[HEADS]);
      System.out.printf("Number of tails: %d%n", frequency[TAILS]);
   }
}
```

```
Number of heads: 46
Number of tails: 54
```

```java
// Fig. 6.7: StudentPoll.java
// Poll analysis program.

public class StudentPoll {
   public static void main(String[] args) {
      // student response array (more typically, input at runtime)
      int[] responses =
         {1, 2, 5, 4, 3, 5, 2, 1, 3, 3, 1, 4, 3, 3, 3, 2, 3, 3, 2, 14};
      int[] frequency = new int[6]; // array of frequency counters

      // for each answer, select responses element and use that value
      // as frequency index to determine array element to increment
      for (int answer = 0; answer < responses.length; ++answer) {
         try {
            ++frequency[responses[answer]];
         }
         catch (ArrayIndexOutOfBoundsException e) {
            System.out.println(e); // invokes toString method
            System.out.printf("   responses[%d] = %d%n%n",
               answer, responses[answer]);
         }
      }

      System.out.printf("%s%10s%n", "Rating", "Frequency");

      // output each array element's value
      for (int rating = 1; rating < frequency.length; ++rating) {
         System.out.printf("%6d%10d%n", rating, frequency[rating]);
      }
   }
}
```

```
java.lang.ArrayIndexOutOfBoundsException: Index 14 out of bounds for length 6
   responses[19] = 14

Rating Frequency
     1         3
     2         4
     3         8
     4         2
     5         2
```

```
1  // Fig. 6.8: EnhancedForTest.java
2  // Using the enhanced for statement to total integers in an array.
3
4  public class EnhancedForTest {
5     public static void main(String[] args) {
6        int[] grades = {87, 68, 94, 100, 83, 78, 85, 91, 76, 87};
7        int total = 0;
8
9        // add each element's value to total
10       for (int grade : grades) {
11          total += grade;
12       }
13
14       System.out.printf("Total of array elements: %d%n", total);
15    }
16 }
```

```
Total of array elements: 849
```

```
for (int counter = 0; counter < grades.length; ++counter) {
   total += grades[counter];
}
```

```
double[] hourlyTemperatures = new double[24];
```

```
processTemps(hourlyTemperatures);
```

```
void processTemps(double[] temps)
```

```
1  // Fig. 6.9: PassArray.java
2  // Passing arrays and individual array elements to methods.
3
4  public class PassArray {
5     // main creates array and calls modifyArray and modifyElement
6     public static void main(String[] args) {
7        int[] array = {1, 2, 3, 4, 5};
8
9        System.out.printf(
10          "Effects of passing reference to entire array:%n" +
11          "The values of the original array are:%n");
12
13       // output original array elements
14       for (int value : array) {
15          System.out.printf("   %d", value);
16       }
17
18       modifyArray(array); // pass array reference
19       System.out.printf("%n%nThe values of the modified array are:%n");
20
21       // output modified array elements
22       for (int value : array) {
23          System.out.printf("   %d", value);
24       }
25
26       System.out.printf(
27          "%n%nEffects of passing array element value:%n" +
28          "array[3] before calling modifyElement: %d%n", array[3]);
```

```
29
30          modifyElement(array[3]); // attempt to modify array[3]
31          System.out.printf(
32             "array[3] after modifyElement completes: %d%n", array[3]);
33       }
34
35       // multiply each element of an array by 2
36       public static void modifyArray(int[] array2) {
37          for (int counter = 0; counter < array2.length; ++counter) {
38             array2[counter] *= 2;
39          }
40       }
41
42       // multiply argument by 2
43       public static void modifyElement(int element) {
44          element *= 2;
45          System.out.printf(
46             "Value of element in modifyElement: %d%n", element);
47       }
48    }
```

```
Effects of passing reference to entire array:
The values of the original array are:
   1   2   3   4   5

The values of the modified array are:
   2   4   6   8   10

Effects of passing array element value:
array[3] before calling modifyElement: 8
Value of element in modifyElement: 16
array[3] after modifyElement completes: 8
```

```
int[][] b = {{1, 2}, {3, 4, 5}};
```

```
int[][] b = new int[2][];   // create 2 rows
b[0] = new int[5]; // create 5 columns for row 0
b[1] = new int[3]; // create 3 columns for row 1
```

```java
// Fig. 6.10: InitArray.java
// Initializing two-dimensional arrays.

public class InitArray {
   // create and output two-dimensional arrays
   public static void main(String[] args) {
      int[][] array1 = {{1, 2, 3}, {4, 5, 6}};
      int[][] array2 = {{1, 2}, {3}, {4, 5, 6}};

      System.out.println("Values in array1 by row are");
      outputArray(array1); // displays array1 by row

      System.out.printf("%nValues in array2 by row are%n");
      outputArray(array2); // displays array2 by row
   }

   // output rows and columns of a two-dimensional array
   public static void outputArray(int[][] array) {
      // loop through array's rows
      for (int row = 0; row < array.length; ++row) {
         // loop through columns of current row
         for (int column = 0; column < array[row].length; ++column) {
            System.out.printf("%d  ", array[row][column]);
         }

         System.out.println();
      }
   }
}
```

```
Values in array1 by row are
1  2  3
4  5  6

Values in array2 by row are
1  2
3
4  5  6
```

```java
// loop through array's rows using enhanced for loop
for (int[] row : array) {
   // loop through columns of current row using enhanced for loop
   for (int element : row) {
      System.out.printf("%d  ", element);
   }

   System.out.println();
}
```

```java
for (int column = 0; column < a[2].length; ++column) {
   a[2][column] = 0;
}
```

```java
// Fig. 6.11: VarargsTest.java
// Using variable-length argument lists.

public class VarargsTest {
   // calculate average
   public static double average(double... numbers) {
      double total = 0.0;

      // calculate total using the enhanced for statement
      for (double d : numbers) {
         total += d;
      }

      return total / numbers.length;
   }

   public static void main(String[] args) {
      double d1 = 10.0;
      double d2 = 20.0;
      double d3 = 30.0;
      double d4 = 40.0;

      System.out.printf("d1 = %.1f%nd2 = %.1f%nd3 = %.1f%nd4 = %.1f%n%n",
         d1, d2, d3, d4);

      System.out.printf("Average of d1 and d2 is %.1f%n",
         average(d1, d2));
      System.out.printf("Average of d1, d2 and d3 is %.1f%n",
         average(d1, d2, d3));
      System.out.printf("Average of d1, d2, d3 and d4 is %.1f%n",
         average(d1, d2, d3, d4));
   }
}
```

```
d1 = 10.0
d2 = 20.0
d3 = 30.0
d4 = 40.0

Average of d1 and d2 is 15.0
Average of d1, d2 and d3 is 20.0
Average of d1, d2, d3 and d4 is 25.0
```

```java
// Fig. 6.12: InitArray.java
// Initializing an array using command-line arguments.

public class InitArray {
   public static void main(String[] args) {
      // check number of command-line arguments
      if (args.length != 3) {
         System.out.println("""
            Error: Please re-execute the program and include
            an array size, initial value and increment.""");
         return;
      }

      // get array size from first command-line argument
      int arrayLength = Integer.parseInt(args[0]);
      int[] array = new int[arrayLength];

      // get initial value and increment from command-line arguments
      int initialValue = Integer.parseInt(args[1]);
      int increment = Integer.parseInt(args[2]);

      // calculate value for each array element
      for (int counter = 0; counter < array.length; ++counter) {
         array[counter] = initialValue + increment * counter;
      }

      System.out.printf("%s%8s%n", "Index", "Value");

```

```
        // display array index and value
        for (int counter = 0; counter < array.length; ++counter) {
           System.out.printf("%5d%8d%n", counter, array[counter]);
        }
     }
  }
```

```
> java InitArray
Error: Please re-execute the program and include
an array size, initial value and increment.""");
```

```
> java InitArray 5 0 4
Index   Value
    0       0
    1       4
    2       8
    3      12
    4      16
```

```
> java InitArray 8 1 2
Index   Value
    0       1
    1       3
    2       5
    3       7
    4       9
    5      11
    6      13
    7      15
```

```
// Fig. 6.13: ArrayManipulations.java
// Arrays class methods and System.arraycopy.
import java.util.Arrays;

public class ArrayManipulations {
   public static void main(String[] args) {
      // sort doubles into ascending order
      double[] doubles = {8.4, 9.3, 0.2, 7.9, 3.4};
      Arrays.sort(doubles);
      System.out.printf("doubles: %s%n", Arrays.toString(doubles));

      // fill 10-element array with 7s
      int[] filledInts = new int[10];
      Arrays.fill(filledInts, 7);
      System.out.printf("filledInts: %s%n", Arrays.toString(filledInts));

      // copy array ints into array intsCopy
      int[] ints = {1, 2, 3, 4, 5, 6};
      int[] intsCopy = new int[ints.length];
      System.arraycopy(ints, 0, intsCopy, 0, ints.length);
      System.out.printf("ints: %s%n", Arrays.toString(ints));
      System.out.printf("intsCopy: %s%n", Arrays.toString(intsCopy));

      // compare ints and intsCopy for equality
      boolean b = Arrays.equals(ints, intsCopy);
      System.out.printf("%nints %s intsCopy%n",
          (b ? "equals" : "does not equal"));

      // compare ints and filledInts for equality
      b = Arrays.equals(ints, filledInts);
      System.out.printf("ints %s filledInts%n",
          (b ? "equals" : "does not equal"));
```

```
        // search ints for the value 5
        int location = Arrays.binarySearch(ints, 5);

        if (location >= 0) {
           System.out.printf("Found 5 at element %d in ints%n", location);
        }
        else {
           System.out.println("5 not found in ints");
        }

        // search intArray for the value 8763
        location = Arrays.binarySearch(ints, 8763);

        if (location >= 0) {
           System.out.printf(
              "Found 8763 at element %d in ints%n", location);
        }
        else {
           System.out.println("8763 not found in ints");
        }
     }
  }
```

```
doubles: [0.2, 3.4, 7.9, 8.4, 9.3]
filledInts: [7, 7, 7, 7, 7, 7, 7, 7, 7, 7]
ints: [1, 2, 3, 4, 5, 6]
intsCopy: [1, 2, 3, 4, 5, 6]

ints equals intsCopy
ints does not equal filledInts
Found 5 at element 4 in ints
8763 not found in ints
```

```
1  // Fig. 6.14: ArrayListCollection.java
2  // Generic ArrayList<E> collection demonstration.
3  import java.util.ArrayList;
4
5  public class ArrayListCollection {
6     public static void main(String[] args) {
7        // create a new ArrayList of Strings with an initial capacity of 3
8        var items = new ArrayList<String>(3);
9        System.out.printf(
10          "After creating list with initial capacity of 3, size is: %d%n",
11          items.size());
12
13       items.add("red"); // append an item to the list
14       items.add(0, "yellow"); // insert "yellow" at index 0
15
16       System.out.print(
17          "Display list contents with counter-controlled for loop:");
18
19       // display the colors in the list
20       for (int i = 0; i < items.size(); ++i) {
21          System.out.printf(" %s", items.get(i));
22       }
23
24       // display colors using enhanced for in the display method
25       display(items,
26          "%nDisplay list contents with enhanced for loop:");
27       System.out.printf("list size: %d%n", items.size());
28
29       items.add("green"); // add "green" to the end of the list
30       items.add("yellow"); // add "yellow" to the end of the list
31       display(items, "\nList with two new elements:");
32       System.out.println("List has grown beyond initial capacity of 3");
33       System.out.printf("list size: %d%n", items.size());
34
35       items.remove("yellow"); // remove the first "yellow"
36       display(items, "\nRemove first instance of yellow:");
37       System.out.printf("list size: %d%n", items.size());
38
```

```
        items.remove(1); // remove item at index 1
        display(items, "\nRemove second list element (green):");
        System.out.printf("list size: %d%n", items.size());

        // check if a value is in the List
        System.out.printf("\n\"red\" is %sin the list%n",
           items.contains("red") ? "": "not ");

        // get the index of an item in the list
        System.out.printf("\"red\" is located at index: %d%n",
           items.indexOf("red"));

        // clear the ArrayList and display its size
        items.clear();
        System.out.printf("\nSize after clear: %s%n", items.size());
     }

     // display the ArrayList's elements on the console
     public static void display(ArrayList<String> items, String header) {
        System.out.printf(header); // display header


        // display each element in items with enhanced for
        for (String item : items) {
           System.out.printf(" %s", item);
        }

        System.out.println();
     }
  }
```

```
After creating list with initial capacity of 3, size is: 0
Display list contents with counter-controlled for loop: yellow red
Display list contents with enhanced for loop: yellow red
list size: 2

List with two new elements: yellow red green yellow
List has grown beyond initial capacity of 3
list size: 4

Remove first instance of yellow: red green yellow
list size: 3

Remove second list element (green): red yellow
list size: 2

"red" is in the list
"red" is located at index: 0

Size after clear: 0
```

```
ArrayList<String> items = new ArrayList<>();
```

```
ArrayList<String> items = new ArrayList<String>();
```

"John Q. Huang"	(a name)
"99999 Main Street"	(a street address)
"Waltham, Massachusetts"	(a city and state)
"(201) 555-1212"	(a telephone number)

```
1  // Fig. 7.1: StringConstructors.java
2  // String class constructors.
3
4  public class StringConstructors {
5     public static void main(String[] args) {
6        char[] charArray = {'b', 'i', 'r', 't', 'h', 'd', 'a', 'y'};
7        var s = new String("hello");
8
9        // use String constructors
10       var s1 = new String(); // ""
11       var s2 = new String(s); // "hello"
12       var s3 = new String(charArray); // "birthday"
13       var s4 = new String(charArray, 5, 3); // "day"
14
15       System.out.printf(
16          "s1 = %s%ns2 = %s%ns3 = %s%ns4 = %s%n", s1, s2, s3, s4);
17    }
18 }
```

```
s1 =
s2 = hello
s3 = birthday
s4 = day
```

```
1  // Fig. 7.2: StringMiscellaneous.java
2  // This program demonstrates the length, charAt and getChars
3  // methods of the String class.
4
5  public class StringMiscellaneous {
6     public static void main(String[] args) {
7        String s1 = "hello there";
8        char[] charArray = new char[5];
9
10       System.out.printf("s1: %s", s1);
11
12       // test length method
13       System.out.printf("%nLength of s1: %d", s1.length());
14
15       // loop through characters in s1 with charAt and display reversed
16       System.out.printf("%nThe string reversed is: ");
17
18       for (int count = s1.length() - 1; count >= 0; --count) {
19          System.out.printf("%c ", s1.charAt(count));
20       }
21
22       // copy characters from s1 into charArray
23       s1.getChars(0, 5, charArray, 0);
24       System.out.printf("%nThe character array is: ");
25
26       for (char character : charArray) {
27          System.out.print(character);
28       }
29
30       System.out.println();
31    }
32 }
```

```
s1: hello there
Length of s1: 11
The string reversed is: e r e h t   o l l e h
The character array is: hello
```

```java
// Fig. 7.3: StringCompare.java
// Comparing Strings and portions of Strings.

public class StringCompare {
   public static void main(String[] args) {
      String s1 = new String("hello"); // s1 is a copy of "hello"
      String s2 = "goodbye";
      String s3 = "Happy Birthday";
      String s4 = "happy birthday";

      System.out.printf(
         "s1 = %s%ns2 = %s%ns3 = %s%ns4 = %s%n%n", s1, s2, s3, s4);

```

```
s1 = hello
s2 = goodbye
s3 = Happy Birthday
s4 = happy birthday
```

```java
      // test for equality
      if (s1.equals("hello")) {  // true
         System.out.println("s1 equals \"hello\"");
      }
      else {
         System.out.println("s1 does not equal \"hello\"");
      }

```

```
s1 equals "hello"
```

```java
      // test for equality with ==
      if (s1 == "hello") { // false; they are not the same object
         System.out.println("s1 is the same object as \"hello\"");
      }
      else {
         System.out.println("s1 is not the same object as \"hello\"");
      }

```

```
s1 is not the same object as "hello"
```

```java
      // test for equality (ignore case)
      if (s3.equalsIgnoreCase(s4)) { // true
         System.out.printf("%s equals %s with case ignored%n", s3, s4);
      }
      else {
         System.out.println("s3 does not equal s4");
      }

```

```
Happy Birthday equals happy birthday with case ignored
```

```
38          // test compareTo
39          System.out.printf(
40             "%ns1.compareTo(s2) is %d", s1.compareTo(s2));
41          System.out.printf(
42             "%ns2.compareTo(s1) is %d", s2.compareTo(s1));
43          System.out.printf(
44             "%ns1.compareTo(s1) is %d", s1.compareTo(s1));
45          System.out.printf(
46             "%ns3.compareTo(s4) is %d", s3.compareTo(s4));
47          System.out.printf(
48             "%ns4.compareTo(s3) is %d%n%n", s4.compareTo(s3));
49
```

```
s1.compareTo(s2) is 1
s2.compareTo(s1) is -1
s1.compareTo(s1) is 0
s3.compareTo(s4) is -32
s4.compareTo(s3) is 32
```

```
50          // test regionMatches (case sensitive)
51          if (s3.regionMatches(0, s4, 0, 5)) {
52             System.out.println("First 5 characters of s3 and s4 match");
53          }
54          else {
55             System.out.println(
56                "First 5 characters of s3 and s4 do not match");
57          }
58
59          // test regionMatches (ignore case)
60          if (s3.regionMatches(true, 0, s4, 0, 5)) {
61             System.out.println(
62                "First 5 characters of s3 and s4 match with case ignored\n");
63          }
64          else {
65             System.out.println(
66                "First 5 characters of s3 and s4 do not match\n");
67          }
68
```

```
First 5 characters of s3 and s4 do not match
First 5 characters of s3 and s4 match with case ignored
```

```
69          // test whether a String contains only whitespace
70          System.out.printf("\" \\t\\n\".isBlank(): %b%n", " \t\n".isBlank());
71       }
72    }
```

```
" \t\n".isBlank(): true
```

```java
// Fig. 7.4: StringStartEnd.java
// String methods startsWith and endsWith.

public class StringStartEnd {
   public static void main(String[] args) {
      String[] strings = {"started", "starting", "ended", "ending"};

      // test method startsWith
      for (String string : strings) {
         if (string.startsWith("st")) {
            System.out.printf("\"%s\" starts with \"st\"%n", string);
         }
      }

      System.out.println();

      // test method startsWith starting from position 2 of string
      for (String string : strings) {
         if (string.startsWith("art", 2)) {
            System.out.printf(
               "\"%s\" starts with \"art\" at position 2%n", string);
         }
      }

      System.out.println();

      // test method endsWith
      for (String string : strings) {
         if (string.endsWith("ed")) {
            System.out.printf("\"%s\" ends with \"ed\"%n", string);
         }
      }
   }
}
```

```
"started" starts with "st"
"starting" starts with "st"

"started" starts with "art" at position 2
"starting" starts with "art" at position 2

"started" ends with "ed"
"ended" ends with "ed"
```

```java
// Fig. 7.5: StringIndexMethods.java
// String searching methods indexOf and lastIndexOf.

public class StringIndexMethods {
   public static void main(String[] args) {
      String letters = "abcdefghijklmabcdefghijklm";

      // test indexOf to locate a character in a String
      System.out.printf(
         "'c' is located at index %d%n", letters.indexOf('c'));
      System.out.printf(
         "'a' is located at index %d%n", letters.indexOf('a', 1));
      System.out.printf(
         "'$' is located at index %d%n%n", letters.indexOf('$'));

```

```
'c' is located at index 2
'a' is located at index 13
'$' is located at index -1
```

```
16         // test lastIndexOf to find a character in a String
17         System.out.printf("Last 'c' is located at index %d%n",
18            letters.lastIndexOf('c'));
19         System.out.printf("Last 'a' is located at index %d%n",
20            letters.lastIndexOf('a', 25));
21         System.out.printf("Last '$' is located at index %d%n%n",
22            letters.lastIndexOf('$'));
23
```

```
Last 'c' is located at index 15
Last 'a' is located at index 13
Last '$' is located at index -1
```

```
24         // test indexOf to locate a substring in a String
25         System.out.printf("\"def\" is located at index %d%n",
26            letters.indexOf("def"));
27         System.out.printf("\"def\" is located at index %d%n",
28            letters.indexOf("def", 7));
29         System.out.printf("\"hello\" is located at index %d%n%n",
30            letters.indexOf("hello"));
31
32         // test lastIndexOf to find a substring in a String
33         System.out.printf("Last \"def\" is located at index %d%n",
34            letters.lastIndexOf("def"));
35         System.out.printf("Last \"def\" is located at index %d%n",
36            letters.lastIndexOf("def", 25));
37         System.out.printf("Last \"hello\" is located at index %d%n",
38            letters.lastIndexOf("hello"));
39      }
40   }
```

```
"def" is located at index 3
"def" is located at index 16
"hello" is located at index -1

Last "def" is located at index 16
Last "def" is located at index 16
Last "hello" is located at index -1
```

```
String s = "to be or not to be";
System.out.printf("First occurrence of \"be\" is at index: %d%n",
   s.indexOf("be"));
System.out.printf("Last occurrence of \"be\" is at index: %d%n",
   s.lastIndexOf("be"));
```

```
First occurrence of "be" is at index: 3
Last occurrence of "be" is at index: 16
```

```
1  // Fig. 7.6: SubString.java
2  // String class substring methods.
3
4  public class SubString {
5     public static void main(String[] args) {
6        String letters = "abcdefghijklmabcdefghijklm";
7
8        // test substring methods
9        System.out.printf("Substring from index 20 to end is \"%s\"%n",
10          letters.substring(20));
11       System.out.printf("%s \"%s\"%n",
12          "Substring from index 3 up to, but not including, 6 is",
13          letters.substring(3, 6));
14    }
15 }
```

```
Substring from index 20 to end is "hijklm"
Substring from index 3 up to, but not including, 6 is "def"
```

```
1  // Fig. 7.7: StringConcatenation.java
2  // String method concat.
3
4  public class StringConcatenation {
5     public static void main(String[] args) {
6        String s1 = "Happy ";
7        String s2 = "Birthday";
8
9        System.out.printf("s1 = %s%ns2 = %s%n%n",s1, s2);
10       System.out.printf(
11          "Result of s1.concat(s2) = %s%n", s1.concat(s2));
12       System.out.printf("s1 after concatenation = %s%n", s1);
13    }
14 }
```

```
s1 = Happy
s2 = Birthday

Result of s1.concat(s2) = Happy Birthday
s1 after concatenation = Happy
```

```
1  // Fig. 7.8: StringMiscellaneous2.java
2  // String methods replace, toLowerCase, toUpperCase, trim and toCharArray.
3
4  public class StringMiscellaneous2 {
5     public static void main(String[] args) {
6        String s1 = "hello";
7        String s2 = "GOODBYE";
8        String s3 = "   spaces   ";
9
10       System.out.printf("s1: \"%s\"%ns2: \"%s\"%ns3: \"%s\"", s1, s2, s3);
11
```

```
s1: "hello"
s2: "GOODBYE"
s3: "   spaces   "
```

```
12       // test method replace
13       System.out.printf(
14          "%n%nReplace 'l' with 'L' in s1: %s%n%n", s1.replace('l', 'L'));
15
```

```
Replace 'l' with 'L' in s1: heLLo
```

```
16       // test toLowerCase and toUpperCase
17       System.out.printf("s1.toUpperCase() = %s%n", s1.toUpperCase());
18       System.out.printf("s2.toLowerCase() = %s%n%n", s2.toLowerCase());
19
```

```
s1 = HELLO
s2 = goodbye
```

```
20       // test strip method
21       System.out.printf("s3 after strip = \"%s\"%n%n", s3.strip());
22
```

```
s3 after strip = "spaces"
```

```
        // test toCharArray method
        char[] charArray = s1.toCharArray();
        System.out.print("s1 as a character array = ");

        for (char character : charArray) {
           System.out.printf("%c ", character);
        }

        System.out.println();
     }
  }
```

```
s1 as a character array = h e l l o
```

```
// Fig. 7.9: StringValueOf.java
// String valueOf methods.

public class StringValueOf {
   public static void main(String[] args) {
      char[] charArray = {'a', 'b', 'c', 'd', 'e', 'f'};
      boolean booleanValue = true;
      char characterValue = 'Z';
      int integerValue = 7;
      long longValue = 10000000000L; // L suffix indicates long
      float floatValue = 2.5f; // f indicates that 2.5 is a float
      double doubleValue = 33.333; // no suffix, double is default
      Object objectRef = "hello"; // assign String to an Object reference

      System.out.printf(
         "char array = %s%n", String.valueOf(charArray));
      System.out.printf("part of char array = %s%n",
         String.valueOf(charArray, 3, 3));
      System.out.printf(
         "boolean = %s%n", String.valueOf(booleanValue));
      System.out.printf(
         "char = %s%n", String.valueOf(characterValue));
      System.out.printf("int = %s%n", String.valueOf(integerValue));
      System.out.printf("long = %s%n", String.valueOf(longValue));
      System.out.printf("float = %s%n", String.valueOf(floatValue));
      System.out.printf(
         "double = %s%n", String.valueOf(doubleValue));
      System.out.printf("Object = %s%n", String.valueOf(objectRef));
   }
}
```

```
char array = abcdef
part of char array = def
boolean = true
char = Z
int = 7
long = 10000000000
float = 2.5
double = 33.333
Object = hello
```

```
// Fig. 7.10: StringBuilderConstructors.java
// StringBuilder constructors.

public class StringBuilderConstructors {
   public static void main(String[] args) {
      var buffer1 = new StringBuilder();
      var buffer2 = new StringBuilder(10);
      var buffer3 = new StringBuilder("hello");

      System.out.printf("buffer1 = \"%s\"%n", buffer1);
      System.out.printf("buffer2 = \"%s\"%n", buffer2);
      System.out.printf("buffer3 = \"%s\"%n", buffer3);
   }
}
```

```
buffer1 = ""
buffer2 = ""
buffer3 = "hello"
```

```java
// Fig. 7.11: StringBuilderCapLen.java
// StringBuilder length, setLength, capacity and ensureCapacity methods.

public class StringBuilderCapLen {
   public static void main(String[] args) {
      var buffer = new StringBuilder("Hello, how are you?");

      System.out.printf("buffer = %s%nlength = %d%ncapacity = %d%n%n",
         buffer.toString(), buffer.length(), buffer.capacity());

      buffer.ensureCapacity(75);
      System.out.printf("New capacity = %d%n%n", buffer.capacity());

      buffer.setLength(10);
      System.out.printf("New length = %d%nbuffer = %s%n",
         buffer.length(), buffer.toString());
   }
}
```

```
buffer = Hello, how are you?
length = 19
capacity = 35

New capacity = 75

New length = 10
buffer = Hello, how
```

```java
// Fig. 7.12: StringBuilderChars.java
// StringBuilder methods charAt, setCharAt, getChars and reverse.

public class StringBuilderChars {
   public static void main(String[] args) {
      var buffer = new StringBuilder("hello there");

      System.out.printf("buffer = %s%n", buffer.toString());
      System.out.printf("Character at 0: %s%nCharacter at 4: %s%n%n",
         buffer.charAt(0), buffer.charAt(4));

      char[] charArray = new char[buffer.length()];
      buffer.getChars(0, buffer.length(), charArray, 0);
      System.out.print("The characters are: ");

      for (char character : charArray) {
         System.out.print(character);
      }

      buffer.setCharAt(0, 'H');
      buffer.setCharAt(6, 'T');
      System.out.printf("%n%nbuffer = %s", buffer.toString());

      buffer.reverse();
      System.out.printf("%n%nbuffer = %s%n", buffer.toString());
   }
}
```

```
buffer = hello there
Character at 0: h
Character at 4: o

The characters are: hello there

buffer = Hello There

buffer = erehT olleH
```

```java
// Fig. 7.13: StringBuilderAppend.java
// StringBuilder append methods.

public class StringBuilderAppend {
   public static void main(String[] args) {
      Object objectRef = "hello";
      String string = "goodbye";
      char[] charArray = {'a', 'b', 'c', 'd', 'e', 'f'};
      boolean booleanValue = true;
      char characterValue = 'Z';
      int integerValue = 7;
      long longValue = 10000000000L;
      float floatValue = 2.5f;
      double doubleValue = 33.333;

      var lastBuffer = new StringBuilder("last buffer");
      var buffer = new StringBuilder();

      buffer.append(objectRef)
            .append(System.lineSeparator())
            .append(string)
            .append(System.lineSeparator())
            .append(charArray)
            .append(System.lineSeparator())
            .append(charArray, 0, 3)
            .append(System.lineSeparator())


            .append(booleanValue)
            .append(System.lineSeparator())
            .append(characterValue)
            .append(System.lineSeparator())
            .append(integerValue)
            .append(System.lineSeparator())
            .append(longValue)
            .append(System.lineSeparator())
            .append(floatValue)
            .append(System.lineSeparator())
            .append(doubleValue)
            .append(System.lineSeparator())
            .append(lastBuffer);

      System.out.printf("buffer contains%n%s%n", buffer.toString());
   }
}
```

```
buffer contains
hello
goodbye
abcdef
abc
true
Z
7
10000000000
2.5
33.333
last buffer
```

```
String s = string1 + string2 + value;
```

```
String s = new StringBuilder().append("hello").append("BC").
   append(22).toString();

s = new StringBuilder().append(s).append("!").toString();
```

```java
// Fig. 7.14: StringBuilderInsertDelete.java
// StringBuilder methods insert, delete and deleteCharAt.

public class StringBuilderInsertDelete {
   public static void main(String[] args) {
      Object objectRef = "hello";
      String string = "goodbye";
      char[] charArray = {'a', 'b', 'c', 'd', 'e', 'f'};
      boolean booleanValue = true;
      char characterValue = 'K';
      int integerValue = 7;
      long longValue = 10000000;
      float floatValue = 2.5f; // f suffix indicates that 2.5 is a float
      double doubleValue = 33.333;

      var buffer = new StringBuilder();

      buffer.insert(0, objectRef);
      buffer.insert(0, "  "); // each of these contains two spaces
      buffer.insert(0, string);
      buffer.insert(0, "  ");
      buffer.insert(0, charArray);
      buffer.insert(0, "  ");
      buffer.insert(0, charArray, 3, 3);
      buffer.insert(0, "  ");
      buffer.insert(0, booleanValue);
      buffer.insert(0, "  ");
      buffer.insert(0, characterValue);


      buffer.insert(0, "  ");
      buffer.insert(0, integerValue);
      buffer.insert(0, "  ");
      buffer.insert(0, longValue);
      buffer.insert(0, "  ");
      buffer.insert(0, floatValue);
      buffer.insert(0, "  ");
      buffer.insert(0, doubleValue);

      System.out.printf(
         "buffer after inserts:%n%s%n%n", buffer.toString());

      buffer.deleteCharAt(10); // delete 5 in 2.5
      buffer.delete(2, 6); // delete .333 in 33.333

      System.out.printf(
         "buffer after deletes:%n%s%n", buffer.toString());
   }
}
```

```
buffer after inserts:
33.333  2.5  10000000  7  K  true  def  abcdef  goodbye  hello

buffer after deletes:
33  2.  10000000  7  K  true  def  abcdef  goodbye  hello
```

```java
// Fig. 7.15: StaticCharMethods.java
// Character static methods for testing characters and converting case.
import java.util.Scanner;

public class StaticCharMethods {
   public static void main(String[] args) {
      var scanner = new Scanner(System.in); // create scanner
      System.out.println("Enter a character and press Enter");
      String input = scanner.next();
      char c = input.charAt(0); // get input character

      // display character info
      System.out.printf("is defined: %b%n", Character.isDefined(c));
      System.out.printf("is digit: %b%n", Character.isDigit(c));
      System.out.printf("is first character in a Java identifier: %b%n",
         Character.isJavaIdentifierStart(c));
      System.out.printf("is part of a Java identifier: %b%n",
         Character.isJavaIdentifierPart(c));
      System.out.printf("is letter: %b%n", Character.isLetter(c));
      System.out.printf(
         "is letter or digit: %b%n", Character.isLetterOrDigit(c));
      System.out.printf(
         "is lower case: %b%n", Character.isLowerCase(c));
      System.out.printf(
         "is upper case: %b%n", Character.isUpperCase(c));
      System.out.printf(
         "to upper case: %c%n", Character.toUpperCase(c));
      System.out.printf(
         "to lower case: %c%n", Character.toLowerCase(c));
   }
}
```

```
Enter a character and press Enter
A
is defined: true
is digit: false
is first character in a Java identifier: true
is part of a Java identifier: true
is letter: true
is letter or digit: true
is lower case: false
is upper case: true
to upper case: A
to lower case: a
```

```
Enter a character and press Enter
8
is defined: true
is digit: true
is first character in a Java identifier: false
is part of a Java identifier: true
is letter: false
is letter or digit: true
is lower case: false
is upper case: false
to upper case: 8
to lower case: 8
```

```
Enter a character and press Enter
$
is defined: true
is digit: false
is first character in a Java identifier: true
is part of a Java identifier: true
is letter: false
is letter or digit: false
is lower case: false
is upper case: false
to upper case: $
to lower case: $
```

```java
// Fig. 7.16: StaticCharMethods2.java
// Character class static conversion methods.
import java.util.Scanner;

public class StaticCharMethods2 {
   public static void main(String[] args) {
      var scanner = new Scanner(System.in);

      // get radix
      System.out.print("Please enter a radix: ");
      int radix = scanner.nextInt();

      // get user choice
      System.out.println("""
         Please choose one:
            1 -- Convert digit to character
            2 -- Convert character to digit""");
      System.out.print(": ");
      int choice = scanner.nextInt();

      // process request
      switch (choice) {
         case 1 -> { // convert digit to character
            System.out.print("Enter a digit: ");
            int digit = scanner.nextInt();
            System.out.printf("Convert digit to character: %s%n",
               Character.forDigit(digit, radix));
         }

         case 2 -> { // convert character to digit
            System.out.print("Enter a character: ");
            char character = scanner.next().charAt(0);
            System.out.printf("Convert character to digit: %s%n",
               Character.digit(character, radix));
         }
      };
   }
}
```

```
Please enter a radix: 16
Please choose one:
   1 -- Convert digit to character
   2 -- Convert character to digit
: 2
Enter a character: A
Convert character to digit: 10
```

```
Please enter a radix: 16
Please choose one:
   1 -- Convert digit to character
   2 -- Convert character to digit
: 1
Enter a digit: 13
Convert digit to character: d
```

```java
// Fig. 7.17: OtherCharMethods.java
// Character class instance methods.
public class OtherCharMethods {
   public static void main(String[] args) {
      Character c1 = 'A';
      Character c2 = 'a';

      System.out.printf(
         "c1 = %c%nc2 = %s%n%n", c1.charValue(), c2.toString());

      if (c1.equals(c2)) {
         System.out.println("c1 and c2 are equal");
      }
      else {
         System.out.println("c1 and c2 are not equal");
      }
   }
}
```

```
c1 = A
c2 = a

c1 and c2 are not equal
```

```java
// Fig. 7.18: TokenTest.java
// Tokenizing with String method split
import java.util.Scanner;

public class TokenTest {
   public static void main(String[] args) {
      // get sentence
      var scanner = new Scanner(System.in);
      System.out.println("Enter a sentence and press Enter");
      String sentence = scanner.nextLine();

      // process user sentence
      String[] tokens = sentence.split(" ");
      System.out.printf("%nNumber of elements: %d%nThe tokens are:%n",
         tokens.length);


      for (String token : tokens) {
         System.out.printf("  %s%n", token);
      }
   }
}
```

```
Enter a sentence and press Enter
This sentence has five tokens

Number of elements: 5
The tokens are:
  This
  sentence
  has
  five
  tokens
```

```java
// Fig. 7.19: RegexExamples.java
// Matching entire Strings to regular expressions.
public class RegexExamples {
   public static void main(String[] args) {
      // fully match a pattern of literal characters
      System.out.println("Matching against: 02215");
      System.out.printf("02215: %b; 51220: %b%n%n",
         "02215".matches("02215"), "51220".matches("02215"));

```

```
Matching against: 02215
02215: true; 51220: false
```

[] {} () \ * + ^ $? . |

```
10          // fully match five digits
11          System.out.println("Matching against: \\d{5}");
12          System.out.printf("02215: %b; 9876: %b%n%n",
13             "02215".matches("\\d{5}"), "9876".matches("\\d{5}"));
14
```

```
Matching against: \d{5}
02215: true; 9876: false
```

```
15          // match a word that starts with a capital letter
16          System.out.println("Matching against: [A-Z][a-z]*");
17          System.out.printf("Angel: %b; tina: %b%n%n",
18             "Angel".matches("[A-Z][a-z]*"), "tina".matches("[A-Z][a-z]*"));
19
```

```
Matching against: [A-Z][a-z]*
Angel: true; tina: false
```

```
20          // match any character that's not a lowercase letter
21          System.out.println("Matching against: [^a-z]");
22          System.out.printf("A: %b; a: %b%n%n",
23             "A".matches("[^a-z]"), "a".matches("[^a-z]"));
24
```

```
Matching against: [^a-z]
A: true; a: false
```

```
25          // match metacharacters as literals in a custom character class
26          System.out.println("Matching against: [*+$]");
27          System.out.printf("*: %b; !: %b%n%n",
28             "*".matches("[*+$]"), "!".matches("[*+$]"));
29
```

```
Matching against: [*+$]
*: true; !: false
```

```
30          // match a capital letter followed by at least one lowercase letter
31          System.out.println("Matching against: [A-Z][a-z]+");
32          System.out.printf("Angel: %b; T: %b%n%n",
33             "Angel".matches("[A-Z][a-z]+"), "T".matches("[A-Z][a-z]+"));
34
```

```
Matching against: [A-Z][a-z]+
Angel: true; T: false
```

```
35          // match zero or one occurrence of a subexpression
36          System.out.println("Matching against: labell?ed");
37          System.out.printf("labelled: %b; labeled: %b; labellled: %b%n%n",
38             "labelled".matches("labell?ed"), "labeled".matches("labell?ed"),
39             "labellled".matches("labell?ed"));
40
```

```
Matching against: labell?ed
labelled: true; labeled: true; labellled: false
```

```
41          // match n (3) or more occurrences of a subexpression
42          System.out.println("Matching against: \\d{3,}");
43          System.out.printf("123: %b; 1234567890: %b; 12: %b%n%n",
44             "123".matches("\\d{3,}"), "1234567890".matches("\\d{3,}"),
45             "12".matches("\\d{3,}"));
46
```

```
Matching against: \d{3,}
123: true; 1234567890: true; 12: false
```

```
        // match n to m inclusive (3-6), occurrences of a subexpression
        System.out.println("Matching against: \\d{3,6}");
        System.out.printf("123: %b; 123456: %b; 1234567: %b; 12: %b%n",
           "123".matches("\\d{3,6}"), "123456".matches("\\d{3,6}"),
           "1234567".matches("\\d{3,6}"), "12".matches("\\d{3,6}"));
     }
  }
```

```
Matching against: \d{3,6}
123: true; 123456: true; 1234567: false; 12: false
```

```
1  // Fig. 7.20: RegexReplacement.java
2  // Regular expression replacements.
3  public class RegexReplacement {
4     public static void main(String[] args) {
5        // replace tabs with commas
6        String s1 = "1\t2\t3\t4";
7        System.out.printf("Original string: %s%n", s1);
8        System.out.printf("New string with commas replacing tabs: %s%n",
9           s1.replaceAll("\\t", ","));
10    }
11 }
```

```
Original string: 1      2       3       4
New string with commas replacing tabs: 1,2,3,4
```

```
1  // Fig. 7.21: RegexMatching.java
2  // Classes Pattern and Matcher.
3  import java.util.regex.Pattern;
4  import java.util.regex.Matcher;
5
6  public class RegexMatching {
7     public static void main(String[] args) {
8        // performing a simple match
9        String s1 = "Programming is fun";
10       System.out.printf("s1: %s%n%n", s1);
11       System.out.println("Search anywhere in s1:");
12
13       Pattern pattern1 = Pattern.compile("Programming");
14       Matcher matcher1 = pattern1.matcher(s1);
15       boolean found1 = matcher1.find();
16
17       Pattern pattern2 = Pattern.compile("fun");
18       Matcher matcher2 = pattern2.matcher(s1);
19       boolean found2 = matcher2.find();
20
21       Pattern pattern3 = Pattern.compile("fn");
22       Matcher matcher3 = pattern3.matcher(s1);
23       boolean found3 = matcher3.find();
24
25       System.out.printf("Programming: %b; fun: %b; fn: %b%n%n",
26          found1, found2, found3);
27
```

```
s1: Programming is fun

Search anywhere in s1:
Programming: true; fun: true; fn: false
```

```
        // ignoring case
        String s2 = "GABRIELA ALVAREZ";
        System.out.printf("s2: %s%n%n", s2);
        System.out.println("Case insensitive search for Gabriela in s2:");

        Pattern pattern4 =
           Pattern.compile("Gabriela", Pattern.CASE_INSENSITIVE);
        Matcher matcher4 = pattern4.matcher(s2);

        if (matcher4.find()) {
           System.out.printf("Gabriela found%n");
           System.out.printf("Matched text: %s%n%n", matcher4.group());
        }
        else {
           System.out.printf("Gabriela not found%n");
        }

```

```
s2: GABRIELA ALVAREZ

Case insensitive search for Gabriela in s2:
Gabriela found
Matched text: GABRIELA
```

```
        // finding all matches
        String contact =
           "Lea Dubois, Home: 555-555-1234, Work: 555-555-4321";
        Pattern phonePattern = Pattern.compile("\\d{3}-\\d{3}-\\d{4}");
        Matcher phoneMatcher = phonePattern.matcher(contact);

        System.out.printf("Finding phone numbers in:%n%s%n", contact);
        while (phoneMatcher.find()) {
           System.out.printf("   %s%n", phoneMatcher.group());
        }
     }
  }
```

```
Finding phone numbers in:
Lea Dubois, Home: 555-555-1234, Work: 555-555-4321
   555-555-1234
   555-555-4321
```

```
  // Fig. 7.22: Wrangling.java
  // Simple data wrangling tasks to prepare text to train language models.
  import java.util.Arrays;

  public class Wrangling {
     public static void main(String[] args) {
        String corpus = """
           In C++, an int  on one  machine  might be 16 bits     (2 bytes)
           of   memory,    on another    32 bits (4 bytes), and on another
           64 bits (8 bytes).    In Java,   int values are  always 32 bits
           (4 bytes). The eight  primitive types in Java -- boolean, byte,
           char, double, float, int, long and short -- are portable across
           all     computer     platforms      that     support      Java.
           """;
        System.out.printf("INITIAL CORPUS:%n%s%n", corpus);

```

```
INITIAL CORPUS:
In C++, an int  on one  machine  might be 16 bits     (2 bytes)
of   memory,    on another    32 bits (4 bytes), and on another
64 bits (8 bytes).    In Java,   int values are  always 32 bits
(4 bytes). The eight  primitive types in Java -- boolean, byte,
char, double, float, int, long and short -- are portable across
all     computer     platforms      that     support      Java.
```

```
        // convert corpus to lowercase
        String lowercased = corpus.toLowerCase();
        System.out.printf("LOWERCASED CORPUS:%n%s%n", lowercased);

```

```
LOWERCASED CORPUS:
in c++, an int  on one  machine  might be 16 bits     (2 bytes)
of   memory,    on another    32 bits (4 bytes), and on another
64 bits (8 bytes).    in java,   int values are  always 32 bits
(4 bytes). the eight  primitive types in java -- boolean, byte,
char, double, float, int, long and short -- are portable across
all     computer     platforms      that     support      java.
```

```
        // remove punctuation
        String noPunctuation = lowercased.replaceAll("\\p{Punct}", "");
        System.out.printf("PUNCTUATION REMOVED:%n%s%n", noPunctuation);

```

```
PUNCTUATION REMOVED:
in c an int  on one  machine  might be 16 bits     2 bytes
of   memory    on another    32 bits 4 bytes and on another
64 bits 8 bytes    in java   int values are  always 32 bits
4 bytes the eight  primitive types in java  boolean byte
char double float int long and short  are portable across
all     computer     platforms      that     support      java
```

```
        // collapsing consecutive whitespace characters to one space
        String spacingNormalized = noPunctuation.replaceAll("\\s+", " ");
        System.out.printf("SPACING NORMALIZED:%n%s%n%n", spacingNormalized);

```

```
SPACING NORMALIZED:
in c an int on one machine might be 16 bits 2 bytes of memory on another 32
bits 4 bytes and on another 64 bits 8 bytes in java int values are always 32
bits 4 bytes the eight primitive types in java boolean byte char double float
int long and short are portable across all computer platforms that support java
```

```
        // tokenizing the text
        String[] tokenized = spacingNormalized.split(" ");
        System.out.printf("TOKENS:%n%s%n", Arrays.toString(tokenized));
    }
}
```

```
TOKENS:
[in, c, an, int, on, one, machine, might, be, 16, bits, 2, bytes, of, memo-
ry, on, another, 32, bits, 4, bytes, and, on, another, 64, bits, 8, bytes,
in, java, int, values, are, always, 32, bits, 4, bytes, the, eight, primi-
tive, types, in, java, boolean, byte, char, double, float, int, long, and,
short, are, portable, across, all, computer, platforms, that, support, java]
```

```
// Fig. 7.23: CryptographyDemo.java
// Encrypting and decrypting messages with AES.
import javax.crypto.Cipher;
import javax.crypto.KeyGenerator;
import javax.crypto.SecretKey;
import java.security.SecureRandom;
import java.util.Base64;
import java.util.Scanner;

public class CryptographyDemo {
```

```
    public static void main(String[] args) {
        var scanner = new Scanner(System.in);

        System.out.println("Enter the text to encrypt:");
        String plaintext = scanner.nextLine();

```

```
Enter the text to encrypt:
Welcome to encryption and decryption using AES in Java!
```

```
17          System.out.println(
18             "\nEnter the secret key seed (for reproducibility):");
19          String seedString = scanner.nextLine();
20
```

```
Enter the secret key seed (for reproducibility):
XMWUJBVYHXZ
```

```
21          try { // encryption/decryption might throw exceptions
22             // generate a reproducible AES key from the seedString
23             SecretKey secretKey = generateKey(seedString);
24             System.out.printf("%nSecretKey:%n%s%n",
25                Base64.getEncoder().encodeToString(secretKey.getEncoded()));
26
```

```
SecretKey:
8ew8jBJVSpkRtG4zorwxd34JBpUxEdOt+ec0QTfr7cE=
```

```
27             // encrypt the plaintext
28             String ciphertext = encrypt(plaintext, secretKey);
29             System.out.printf("%nEncrypted:%n%s%n", ciphertext);
30
```

```
Encrypted:
Cz9o3YoFcX+yebfAOuMPMi+UZz2bc5jJdstfzapaNKwJldZuQH1f3QKNKrh9Zh4aB5tRbns6ZL9b
XF8Bq2YNeQ==
```

```
31             // decrypt the text
32             System.out.printf("%nDecrypted:%n%s%n",
33                decrypt(ciphertext, secretKey));
34
```

```
Decrypted:
Welcome to encryption and decryption using AES in Java!
```

```
35             // decrypt ciphertext entered by the user
36             System.out.println("\nEnter the ciphertext to decipher:");
37             ciphertext = scanner.nextLine();
38             System.out.printf("%nDecrypted:%n%s%n",
39                decrypt(ciphertext, secretKey));
40          }
41          catch (Exception ex) {
42             System.out.printf("""
43                An exception occurred during encryption/decryption:
44                %s%n
45                """, ex);
46          }
47       }
48
```

```
Enter the ciphertext to decipher:
pMa5tfEKwk59dTvCO4Ft1IFQz9mEXnkfYXZwxk4ujGE=

Decrypted:
Symmetric-Key Cryptography
```

```
49       // Generate and return a SecretKey. Note: We are seeding a
50       // SecureRandom object for reproducibility so you can decipher this
51       // chapter's encrypted text. Don't do this in production code.
52       public static SecretKey generateKey(String seed) throws Exception {
53          // get a secure random number generator
54          var random = SecureRandom.getInstance("SHA1PRNG");
55          random.setSeed(seed.getBytes()); // for demo reproducibility
56
57          // get an AES SecretKey generator
58          var keyGen = KeyGenerator.getInstance("AES");
59          keyGen.init(256, random); // set up for 256-bit keys
60          return keyGen.generateKey(); // generate and return new SecretKey
61       }
62
```

```
63      // encrypt a String using AES
64      public static String encrypt(
65         String plaintext, SecretKey secretKey) throws Exception {
66
67         var cipher = Cipher.getInstance("AES"); // get a Cipher object
68         cipher.init(Cipher.ENCRYPT_MODE, secretKey); // configure to encrypt
69
70         // encrypt the byte[] representation of plaintext
71         byte[] encrypted = cipher.doFinal(plaintext.getBytes());
72
73         // Base64 encode encrypted as a ciphertext String and return
74         return Base64.getEncoder().encodeToString(encrypted);
75      }
76
```

```
77      // decrypt an AES-encrypted String
78      public static String decrypt(
79         String ciphertext, SecretKey secretKey) throws Exception {
80
81         var cipher = Cipher.getInstance("AES"); // get a Cipher object
82         cipher.init(Cipher.DECRYPT_MODE, secretKey); // configure to decrypt
83
84         // Base64 decode ciphertext to byte array
85         byte[] ciphertextBytes = Base64.getDecoder().decode(ciphertext);
86
87         // create plaintext String from ciphertextBytes and return it
88         return new String(cipher.doFinal(ciphertextBytes));
89      }
90   }
```

```java
// Fig. 8.1: Account.java
// Account class with a name instance variable
// and methods to set and get its value.

public class Account {
   private String name; // instance variable

   // method to set the name
   public void setName(String name) {
      this.name = name; // store the name
   }

   // method to retrieve the name
   public String getName() {
      return name; // return value of name to caller
   }
}
```

```
private String name; // instance variable
```

```
Error: Main method not found in class Account, please define the
main method as:
   public static void main(String[] args)
```

```java
// Fig. 8.2: AccountTest.java
// Creating and manipulating an Account object.
import java.util.Scanner;

public class AccountTest {
   public static void main(String[] args) {
      // create a Scanner to obtain input from the user
      var input = new Scanner(System.in);

      // create an Account object and assign it to myAccount
      var myAccount = new Account();

      // display initial value of name (null)
      System.out.printf("Initial name is: %s%n%n", myAccount.getName());

      // prompt for and read name
      System.out.println("Please enter the name:");
      String theName = input.nextLine(); // read a line of text
      myAccount.setName(theName); // put theName in myAccount
      System.out.println(); // outputs a blank line

      // display the name stored in object myAccount
      System.out.printf("Name in object myAccount is:%n%s%n",
         myAccount.getName());
   }
}
```

```
Initial name is: null

Please enter the name:
Sofia Lopez

Name in object myAccount is:
Sofia Lopez
```

```
javac Account.java AccountTest.java
```

```
var input = new java.util.Scanner(System.in);
```

```
private int numberOfStudents = 10;
```

```
var account1 = new Account("Wei Li");
```

```
1  // Fig. 8.3: Account.java
2  // Account class with a constructor that initializes the name.
3
4  public class Account {
5     private String name; // instance variable
6
7     // constructor initializes name with parameter name
8     public Account(String name) { // constructor name is class name
9        this.name = name;
10    }
11
12    // method to set the name
13    public void setName(String name) {
14       this.name = name;
15    }
16
17    // method to retrieve the name
18    public String getName() {
19       return name;
20    }
21 }
```

```
1  // Fig. 8.4: AccountTest.java
2  // Using the Account constructor to initialize the name
3  // instance variable of each new Account object.
4
5  public class AccountTest {
6     public static void main(String[] args) {
7        // create two Account objects
8        var account1 = new Account("Wei Li");
9        var account2 = new Account("Logan Brown");
10
11       // display initial value of name for each Account
12       System.out.printf("account1 name is: %s%n", account1.getName());
13       System.out.printf("account2 name is: %s%n", account2.getName());
14    }
15 }
```

```
account1 name is: Wei Li
account2 name is: Logan Brown
```

```
var myAccount = new Account();
```

```
1  // Fig. 8.5: Account.java
2  // Account class with a BigDecimal instance variable balance, and
3  // a constructor and deposit method that perform validation.
4  import java.math.BigDecimal;
5
6  public class Account {
7     private String name; // instance variable
8     private BigDecimal balance = BigDecimal.ZERO; // instance variable
9
10    // Account constructor that receives two parameters
11    public Account(String name, BigDecimal balance) {
12       this.name = name; // assign name to instance variable name
13
14       // validate that the balance is greater than 0; if it's not,
15       // instance variable balance keeps its default initial value of 0
16       if (balance.compareTo(BigDecimal.ZERO) > 0) { // is balance valid?
17          this.balance = balance; // assign it to instance variable balance
18       }
19    }
```

```
20
21      // method that deposits (adds) only a valid amount to the balance
22      public void deposit(BigDecimal amount) {
23         if (amount.compareTo(BigDecimal.ZERO) > 0) { // is amount valid?
24            balance = balance.add(amount); // add it to the balance
25         }
26      }
27
28      // method returns the account balance
29      public BigDecimal getBalance() {
30         return balance;
31      }
32
33      // method that sets the name
34      public void setName(String name) {
35         this.name = name;
36      }
37
38      // method that returns the name
39      public String getName() {
40         return name;
41      }
42   }
```

```
1   // Fig. 8.6: AccountTest.java
2   // Inputting and outputting floating-point numbers with Account objects.
3   import java.math.BigDecimal;
4   import java.math.RoundingMode;
5   import java.util.Scanner;
6
7   public class AccountTest {
8      public static void main(String[] args) {
9         var account1 = new Account("Wei Li", new BigDecimal("50.00"));
10        var account2 = new Account("Logan Brown", new BigDecimal("-7.53"));
11
12        // display initial balance of each object
13        System.out.printf("%s balance: $%s%n", account1.getName(),
14           account1.getBalance().setScale(2, RoundingMode.HALF_EVEN));
15        System.out.printf("%s balance: $%s%n%n", account2.getName(),
16           account2.getBalance().setScale(2, RoundingMode.HALF_EVEN));
17
18        // create a Scanner to obtain input from the user
19        var input = new Scanner(System.in);
20
21        System.out.print("Enter deposit amount for account1: "); // prompt
22        var depositAmount = input.nextBigDecimal(); // get input
23        System.out.printf("%nadding %s to account1 balance%n%n",
24           depositAmount.setScale(2, RoundingMode.HALF_EVEN));
25        account1.deposit(depositAmount); // add to account1 balance
26
27        // display balances
28        System.out.printf("%s balance: $%s%n", account1.getName(),
29           account1.getBalance().setScale(2, RoundingMode.HALF_EVEN));
30        System.out.printf("%s balance: $%s%n%n", account2.getName(),
31           account2.getBalance().setScale(2, RoundingMode.HALF_EVEN));
```

```
32
33          System.out.print("Enter deposit amount for account2: "); // prompt
34          depositAmount = input.nextBigDecimal(); // get input
35          System.out.printf("%nadding %s to account2 balance%n%n",
36             depositAmount.setScale(2, RoundingMode.HALF_EVEN));
37          account2.deposit(depositAmount); // add to account2 balance
38
39          // display balances
40          System.out.printf("%s balance: $%s%n", account1.getName(),
41             account1.getBalance().setScale(2, RoundingMode.HALF_EVEN));
42          System.out.printf("%s balance: $%s%n%n", account2.getName(),
43             account2.getBalance().setScale(2, RoundingMode.HALF_EVEN));
44       }
45    }
```

```
Wei Li balance: $50.00
Logan Brown balance: $0.00

Enter deposit amount for account1: 25.53

adding 25.53 to account1 balance

Wei Li balance: $75.53
Logan Brown balance: $0.00

Enter deposit amount for account2: 123.45

adding 123.45 to account2 balance

Wei Li balance: $75.53
Logan Brown balance: $123.45
```

```
1  // Fig. 8.7: Card.java
2  // Card class represents a playing card.
3
4  public class Card {
5     private final String face; // face of card ("Ace", "Deuce", ...)
6     private final String suit; // suit of card ("Hearts", "Diamonds", ...)
7
8     // two-argument constructor initializes card's face and suit
9     public Card(String face, String suit) {
10       this.face = face; // initialize face of card
11       this.suit = suit; // initialize suit of card
12    }
13
14    // return String representation of Card
15    public String toString() {
16       return face + " of " + suit;
17    }
18 }
```

```
1  // Fig. 8.8: DeckOfCards.java
2  // DeckOfCards class represents a deck of playing cards.
3  import java.util.random.RandomGenerator;
4
5  public class DeckOfCards {
6     // random number generator
7     private static final RandomGenerator randomNumbers =
8        RandomGenerator.getDefault();
9     public static final int NUMBER_OF_CARDS = 52; // constant # of Cards
10    private Card[] deck = new Card[NUMBER_OF_CARDS]; // Card references
11    private int currentCardIndex = 0; // index of next Card to deal (0-51)
12
```

```
13    // constructor fills deck of Cards
14    public DeckOfCards() {
15       String[] faces = {"Ace", "Deuce", "Three", "Four", "Five", "Six",
16          "Seven", "Eight", "Nine", "Ten", "Jack", "Queen", "King"};
17       String[] suits = {"Hearts", "Diamonds", "Clubs", "Spades"};
18
19       // populate deck with Card objects
20       for (int count = 0; count < deck.length; ++count) {
21          deck[count] = new Card(faces[count % 13], suits[count / 13]);
22       }
23    }
24
```

```
25      // shuffle deck of Cards with one-pass algorithm
26      public void shuffle() {
27         // next call to method dealCard should start at deck[0] again
28         currentCardIndex = 0;
29
30         // for each Card, pick another random Card (0-51) and swap them
31         for (int first = 0; first < deck.length; ++first) {
32            // select a random number between 0 and 51
33            int second = randomNumbers.nextInt(NUMBER_OF_CARDS);
34
35            // swap current Card with randomly selected Card
36            Card temp = deck[first];
37            deck[first] = deck[second];
38            deck[second] = temp;
39         }
40      }
41
```

```
42      // deal one Card
43      public Card dealCard() {
44         // determine whether Cards remain to be dealt
45         if (currentCardIndex < deck.length) {
46            return deck[currentCardIndex++]; // return current Card in array
47         }
48         else {
49            return null; // return null to indicate that all Cards were dealt
50         }
51      }
52   } // end class DeckOfCards
```

```
1   // Fig. 8.9: DeckOfCardsTest.java
2   // Card shuffling and dealing.
3
4   public class DeckOfCardsTest {
5      // execute program
6      public static void main(String[] args) {
7         var myDeckOfCards = new DeckOfCards();
8         myDeckOfCards.shuffle(); // place Cards in random order
9
10        // print all 52 Cards in the order in which they are dealt
11        for (int i = 1; i <= DeckOfCards.NUMBER_OF_CARDS; ++i) {
12           // deal and display a Card
13           System.out.printf("%-19s", myDeckOfCards.dealCard());
14
15           if (i % 4 == 0) { // output a newline after every fourth card
16              System.out.println();
17           }
18        }
19     }
20  }
```

```
Six of Spades       Eight of Spades     Six of Clubs        Nine of Hearts
Queen of Hearts     Seven of Clubs      Nine of Spades      King of Hearts
Three of Diamonds   Deuce of Clubs      Ace of Hearts       Ten of Spades
Four of Spades      Ace of Clubs        Seven of Diamonds   Four of Hearts
Three of Clubs      Deuce of Hearts     Five of Spades      Jack of Diamonds
King of Clubs       Ten of Hearts       Three of Hearts     Six of Diamonds
Queen of Clubs      Eight of Diamonds   Deuce of Diamonds   Ten of Diamonds
Three of Spades     King of Diamonds    Nine of Clubs       Six of Hearts
Ace of Spades       Four of Diamonds    Seven of Hearts     Eight of Clubs
Deuce of Spades     Eight of Hearts     Five of Hearts      Queen of Spades
Jack of Hearts      Seven of Spades     Four of Clubs       Nine of Diamonds
Ace of Diamonds     Queen of Diamonds   Five of Clubs       King of Spades
Five of Diamonds    Ten of Clubs        Jack of Spades      Jack of Clubs
```

```java
// Fig. 8.10: Time1.java
// Time1 class declaration maintains the time in 24-hour format.

public class Time1 {
   private int hour; // 0 - 23
   private int minute; // 0 - 59
   private int second; // 0 - 59

   // set a new time value using universal time; throw an
   // exception if the hour, minute or second is invalid
   public void setTime(int hour, int minute, int second) {
      // validate hour, minute and second
      if (hour < 0 || hour >= 24 || minute < 0 || minute >= 60 ||
         second < 0 || second >= 60) {
         throw new IllegalArgumentException(
            "hour, minute and/or second was out of range");
      }

      this.hour = hour;
      this.minute = minute;
      this.second = second;
   }

   // convert to String in universal-time format (HH:MM:SS)
   public String toUniversalString() {
      return String.format("%02d:%02d:%02d", hour, minute, second);
   }

   // convert to String in standard-time format (H:MM:SS AM or PM)
   public String toString() {
      return String.format("%d:%02d:%02d %s",
         ((hour == 0 || hour == 12) ? 12 : hour % 12),
         minute, second, (hour < 12 ? "AM" : "PM"));
   }
}
```

```java
// Fig. 8.11: Time1Test.java
// Time1 object used in an app.

public class Time1Test {
   public static void main(String[] args) {
      // create and initialize a Time1 object
      var time = new Time1(); // invokes Time1 constructor

      // output string representations of the time
      displayTime("After time object is created", time);
      System.out.println();

      // change time and output updated time
      time.setTime(13, 27, 6);
      displayTime("After calling setTime", time);
      System.out.println();

      // attempt to set time with invalid values
      try {
         time.setTime(99, 99, 99); // all values out of range
      }
      catch (IllegalArgumentException e) {
         System.out.printf("Exception: %s%n%n", e.getMessage());
      }
```

```
25
26          // display time after attempt to set invalid values
27          displayTime("After calling setTime with invalid values", time);
28       }
29
30       // displays a Time1 object in 24-hour and 12-hour formats
31       private static void displayTime(String header, Time1 t) {
32          System.out.printf("%s%nUniversal time: %s%nStandard time: %s%n",
33             header, t.toUniversalString(), t.toString());
34       }
35    }
```

```
After time object is created
Universal time: 00:00:00
Standard time: 12:00:00 AM

After calling setTime
Universal time: 13:27:06
Standard time: 1:27:06 PM

Exception: hour, minute and/or second was out of range

After calling setTime with invalid values
Universal time: 13:27:06
Standard time: 1:27:06 PM
```

```
1  // Fig. 8.12: MemberAccessTest.java
2  // Private members of class Time1 are not accessible.
3  public class MemberAccessTest {
4     public static void main(String[] args) {
5        var time = new Time1(); // create and initialize Time1 object
6
7        time.hour = 7; // error: hour has private access in Time1
8        time.minute = 15; // error: minute has private access in Time1
9        time.second = 30; // error: second has private access in Time1
10    }
11 }
```

```
.\MemberAccessTest.java:7: error: hour has private access in Time1
      time.hour = 7; // error: hour has private access in Time1
           ^
.\MemberAccessTest.java:8: error: minute has private access in Time1
      time.minute = 15; // error: minute has private access in Time1
           ^
.\MemberAccessTest.java:9: error: second has private access in Time1
      time.second = 30; // error: second has private access in Time1
           ^
3 errors
error: compilation failed
```

```java
// Fig. 8.13: ThisTest.java
// this used implicitly and explicitly to refer to members of an object.

public class ThisTest {
   public static void main(String[] args) {
      var time = new SimpleTime(15, 30, 19);
      System.out.println(time.buildString());
   }
}

// class SimpleTime demonstrates the "this" reference
class SimpleTime {
   private int hour; // 0-23
   private int minute; // 0-59
   private int second; // 0-59

   // if the constructor uses parameter names identical to
   // instance variable names the "this" reference is
   // required to distinguish between the names
   public SimpleTime(int hour, int minute, int second) {
      this.hour = hour; // set "this" object's hour
      this.minute = minute; // set "this" object's minute
      this.second = second; // set "this" object's second
   }

   // use explicit and implicit "this" to call toUniversalString
   public String buildString() {
      return String.format("%24s: %s%n%24s: %s",
         "this.toUniversalString()", this.toUniversalString(),
         "toUniversalString()", toUniversalString());
   }

   // convert to String in universal-time format (HH:MM:SS)
   public String toUniversalString() {
      // "this." is not required here to access instance variables,
      // because this method does not have local variables with same
      // names as instance variables
      return String.format("%02d:%02d:%02d",
         this.hour, this.minute, this.second);
   }
}
```

```
this.toUniversalString(): 15:30:19
     toUniversalString(): 15:30:19
```

```java
// Fig. 8.14: Time2.java
// Time2 class declaration with overloaded constructors.

public class Time2 {
   private int hour; // 0 - 23
   private int minute; // 0 - 59
   private int second; // 0 - 59

   // Time2 no-argument constructor:
   // initializes each instance variable to zero
   public Time2() {
      this(0, 0, 0); // delegate to three-argument constructor
   }

   // Time2 constructor: hour supplied, minute and second defaulted to 0
   public Time2(int hour) {
      this(hour, 0, 0); // delegate to three-argument constructor
   }

   // Time2 constructor: hour and minute supplied, second defaulted to 0
   public Time2(int hour, int minute) {
      this(hour, minute, 0); // delegate to three-argument constructor
   }
```

```
25      // Time2 constructor: hour, minute and second supplied
26      public Time2(int hour, int minute, int second) {
27         if (hour < 0 || hour >= 24) {
28            throw new IllegalArgumentException("hour must be 0-23");
29         }
30
31         if (minute < 0 || minute >= 60) {
32            throw new IllegalArgumentException("minute must be 0-59");
33         }
34
35         if (second < 0 || second >= 60) {
36            throw new IllegalArgumentException("second must be 0-59");
37         }
38
39         this.hour = hour;
40         this.minute = minute;
41         this.second = second;
42      }
43
```

```
44      // Time2 constructor: another Time2 object supplied
45      public Time2(Time2 time) {
46         // delegate to three-argument constructor
47         this(time.hour, time.minute, time.second);
48      }
49
```

```
50      // Set Methods
51      // set a new time value using universal time;
52      // validate the data
53      public void setTime(int hour, int minute, int second) {
54         if (hour < 0 || hour >= 24) {
55            throw new IllegalArgumentException("hour must be 0-23");
56         }
57
58         if (minute < 0 || minute >= 60) {
59            throw new IllegalArgumentException("minute must be 0-59");
60         }
61
62         if (second < 0 || second >= 60) {
63            throw new IllegalArgumentException("second must be 0-59");
64         }
65
66         this.hour = hour;
67         this.minute = minute;
68         this.second = second;
69      }
70
```

```
71      // validate and set hour
72      public void setHour(int hour) {
73         if (hour < 0 || hour >= 24) {
74            throw new IllegalArgumentException("hour must be 0-23");
75         }
76
77         this.hour = hour;
78      }
79
80      // validate and set minute
81      public void setMinute(int minute) {
82         if (minute < 0 || minute >= 60) {
83            throw new IllegalArgumentException("minute must be 0-59");
84         }
85
86         this.minute = minute;
87      }
88
89      // validate and set second
90      public void setSecond(int second) {
91         if (second < 0 || second >= 60) {
92            throw new IllegalArgumentException("second must be 0-59");
93         }
94
95         this.second = second;
96      }
97
```

```
98      // Get Methods
99      // get hour value
100     public int getHour() {return hour;}
101
102     // get minute value
103     public int getMinute() {return minute;}
104
105     // get second value
106     public int getSecond() {return second;}
107
```

```
108     // convert to String in universal-time format (HH:MM:SS)
109     public String toUniversalString() {
110        return String.format("%02d:%02d:%02d",
111           getHour(), getMinute(), getSecond());
112     }
113
114     // convert to String in standard-time format (H:MM:SS AM or PM)
115     public String toString() {
116        return String.format("%d:%02d:%02d %s",
117           ((getHour() == 0 || getHour() == 12) ? 12 : getHour() % 12),
118           getMinute(), getSecond(), (getHour() < 12 ? "AM" : "PM"));
119     }
120  }
```

```java
// Fig. 8.15: Time2Test.java
// Overloaded constructors used to initialize Time2 objects.

public class Time2Test {
   public static void main(String[] args) {
      var t1 = new Time2(); // 00:00:00
      var t2 = new Time2(2); // 02:00:00
      var t3 = new Time2(21, 34); // 21:34:00
      var t4 = new Time2(12, 25, 42); // 12:25:42
      var t5 = new Time2(t4); // 12:25:42

      System.out.println("Constructed with:");
      displayTime("t1: all default arguments", t1);
      displayTime("t2: hour specified; default minute and second", t2);
      displayTime("t3: hour and minute specified; default second", t3);
      displayTime("t4: hour, minute and second specified", t4);
      displayTime("t5: Time2 object t4 specified", t5);

      // attempt to initialize t6 with invalid values
      try {
         var t6 = new Time2(27, 74, 99); // invalid values
      }
      catch (IllegalArgumentException e) {
         System.out.printf("%nException while initializing t6: %s%n",
            e.getMessage());
      }
   }


   // displays a Time2 object in 24-hour and 12-hour formats
   private static void displayTime(String header, Time2 t) {
      System.out.printf("%s%n   %s%n   %s%n",
         header, t.toUniversalString(), t.toString());
   }
}
```

```
Constructed with:
t1: all default arguments
   00:00:00
   12:00:00 AM
t2: hour specified; default minute and second
   02:00:00
   2:00:00 AM
t3: hour and minute specified; default second
   21:34:00
   9:34:00 PM
t4: hour, minute and second specified
   12:25:42
   12:25:42 PM
t5: Time2 object t4 specified
   12:25:42
   12:25:42 PM

Exception while initializing t6: hour must be 0-23
```

```java
public Time2(int hour, int minute)
```

```java
public Time2(int hour, int minute, int second)
```

```java
// Fig. 8.16: Date.java
// Date class declaration.

public class Date {
   private int month; // 1-12
   private int day; // 1-31 based on month
   private int year; // any year

   private static final int[] daysPerMonth =
      {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};

   // constructor: confirm proper value for month and day given the year
   public Date(int month, int day, int year) {
      // check if month in range
      if (month <= 0 || month > 12) {
         throw new IllegalArgumentException(
            "month (" + month + ") must be 1-12");
      }

      // check if day in range for month
      if (day <= 0 ||
         (day > daysPerMonth[month] && !(month == 2 && day == 29))) {
         throw new IllegalArgumentException("day (" + day +
            ") out-of-range for the specified month and year");
      }

      // check for leap year if month is 2 and day is 29
      if (month == 2 && day == 29 && !(year % 400 == 0 ||
           (year % 4 == 0 && year % 100 != 0))) {
         throw new IllegalArgumentException("day (" + day +
            ") out-of-range for the specified month and year");
      }

      this.month = month;
      this.day = day;
      this.year = year;

      System.out.printf("Date object constructor for date %s%n", this);
   }

   // return a String of the form month/day/year
   public String toString() {
      return String.format("%d/%d/%d", month, day, year);
   }
}
```

```java
// Fig. 8.17: Employee.java
// Employee class with references to other objects.

public class Employee {
   private String firstName;
   private String lastName;
   private Date birthDate;
   private Date hireDate;

   // constructor to initialize name, birth date and hire date
   public Employee(String firstName, String lastName, Date birthDate,
      Date hireDate) {
      this.firstName = firstName;
      this.lastName = lastName;
      this.birthDate = birthDate;
      this.hireDate = hireDate;
   }

   // convert Employee to String format
   public String toString() {
      return String.format("%s, %s  Hired: %s  Birthday: %s",
         lastName, firstName, hireDate, birthDate);
   }
}
```

```java
// Fig. 8.18: EmployeeTest.java
// Composition demonstration.

public class EmployeeTest {
   public static void main(String[] args) {
      var birth = new Date(7, 24, 1949);
      var hire = new Date(3, 12, 1988);
      var employee = new Employee("James", "Ramos", birth, hire);

      System.out.println(employee);
   }
}
```

```
Date object constructor for date 7/24/1949
Date object constructor for date 3/12/1988
Ramos, James  Hired: 3/12/1988  Birthday: 7/24/1949
```

```java
// Fig. 8.19: Book.java
// Declaring an enum type with a constructor and explicit instance fields
// and accessors for these fields

public enum Book {
   // declare constants of enum type
   JHTP("Java How to Program", "2025"),
   CHTP("C How to Program", "2020"),
   CPP20FP("C++20 for Programmers", "2023"),
   CPPHTP("C++ How to Program", "2024"),
   PYCDS("Intro to Python for Computer Science and Data Science", "2020"),
   PYFP("Python for Programmers", "2020");

   // instance fields
   private final String title; // book title
   private final String copyrightYear; // copyright year

   // enum constructor
   Book(String title, String copyrightYear) {
      this.title = title;
      this.copyrightYear = copyrightYear;
   }

   // accessor for field title
   public String getTitle() {
      return title;
   }

   // accessor for field copyrightYear
   public String getCopyrightYear() {
      return copyrightYear;
   }
}
```

```java
// Fig. 8.20: EnumTest.java
// Testing enum type Book.
import java.util.EnumSet;

public class EnumTest {
   public static void main(String[] args) {
      System.out.println("All books:");

      // print all books in enum Book
      for (Book book : Book.values()) {
         System.out.printf("%-10s%-57s%s%n", book,
             book.getTitle(), book.getCopyrightYear());
      }

      System.out.printf("%nDisplay a range of enum constants:%n");

      // print first four books
      for (Book book : EnumSet.range(Book.JHTP, Book.CPPHTP)) {
         System.out.printf("%-10s%-57s%s%n", book,
             book.getTitle(), book.getCopyrightYear());
      }
   }
}
```

```
All books:
JHTP      Java How to Program                                       2025
CHTP      C How to Program                                          2020
CPP20FP   C++20 for Programmers                                     2023
CPPHTP    C++ How to Program                                        2024
PYCDS     Intro to Python for Computer Science and Data Science     2020
PYFP      Python for Programmers                                    2020

Display a range of enum constants:
JHTP      Java How to Program                                       2025
CHTP      C How to Program                                          2020
CPP20FP   C++20 for Programmers                                     2023
CPPHTP    C++ How to Program                                        2024
```

```java
// Fig. 8.21: Employee.java
// static variable used to maintain a count of the number of
// Employee objects in memory.

public class Employee {
   private static int count = 0; // number of Employees created
   private String firstName;
   private String lastName;

   // initialize Employee, add 1 to static count and output
   // a String indicating that the constructor was called
   public Employee(String firstName, String lastName) {
      this.firstName = firstName;
      this.lastName = lastName;

      ++count; // increment static count of employees
      System.out.printf("Employee constructor: %s %s; count = %d%n",
         firstName, lastName, count);
   }

   // get first name
   public String getFirstName() {
      return firstName;
   }

   // get last name
   public String getLastName() {
      return lastName;
   }

   // static method to get static count value
   public static int getCount() {
      return count;
   }
}
```

```java
// Fig. 8.22: EmployeeTest.java
// static member demonstration.

public class EmployeeTest {
   public static void main(String[] args) {
      // show that count is 0 before creating Employees
      System.out.printf("Employees before instantiation: %d%n",
         Employee.getCount());

      // create two Employees; count should then be 2
      var e1 = new Employee("Anna", "Gruber");
      var e2 = new Employee("Teva", "Maihi");

      // show that count is 2 after creating two Employees
      System.out.printf("%nEmployees after instantiation:%n");
      System.out.printf("via e1.getCount(): %d%n", e1.getCount());
      System.out.printf("via e2.getCount(): %d%n", e2.getCount());
      System.out.printf("via Employee.getCount(): %d%n",
         Employee.getCount());

      // get names of Employees
      System.out.printf("%nEmployee 1: %s %s%nEmployee 2: %s %s%n",
         e1.getFirstName(), e1.getLastName(),
         e2.getFirstName(), e2.getLastName());
   }
}
```

```
Employees before instantiation: 0
Employee constructor: Anna Gruber; count = 1
Employee constructor: Teva Maihi; count = 2

Employees after instantiation:
via e1.getCount(): 2
via e2.getCount(): 2
via Employee.getCount(): 2

Employee 1: Anna Gruber
Employee 2: Teva Maihi
```

```
import static packageName.ClassName.staticMemberName;
```

```
import static packageName.ClassName.*;
```

```java
// Fig. 8.23: StaticImportTest.java
// Static import of Math class methods.
import static java.lang.Math.*;

public class StaticImportTest {
   public static void main(String[] args) {
      System.out.printf("sqrt(900.0) = %.1f%n", sqrt(900.0));
      System.out.printf("ceil(-9.8) = %.1f%n", ceil(-9.8));
      System.out.printf("E = %f%n", E);
      System.out.printf("PI = %f%n", PI);
   }
}
```

```
sqrt(900.0) = 30.0
ceil(-9.8) = -9.0
E = 2.718282
PI = 3.141593
```

```
// Fig. 8.24: PackageDataTest.java
// Package-access members of a class are accessible by other classes
// in the same package.

public class PackageDataTest {
   public static void main(String[] args) {
      var packageData = new PackageData();

      // output String representation of packageData
      System.out.printf("After instantiation:%n%s%n", packageData);

      // change package access data in packageData object
      packageData.number = 77;
      packageData.string = "Goodbye";

      // output String representation of packageData
      System.out.printf("%nAfter changing values:%n%s%n", packageData);
   }
}

// class with package access instance variables
class PackageData {
   int number = 0; // package-access instance variable
   String string = "Hello"; // package-access instance variable

   // return PackageData object String representation
   public String toString() {
      return String.format("number: %d; string: %s", number, string);
   }
}
```

```
After instantiation:
number: 0; string: Hello

After changing values:
number: 77; string: Goodbye
```

TypeName[*variable1*=*value1*, *variable2*=*value2*, ...]

```
// Fig. 8.25: Records.java
// Demonstrating an Employee record class.
record Employee(
   String first,
   String last,
   double salary,
   String department) {

   // compact constructor for validation
   Employee {
      if (salary < 0.0) {
         throw new IllegalArgumentException("salary must be >= 0.0");
      }
   }
}

public class Records {
   public static void main(String[] args) {
      Employee e1 = new Employee("Prince", "Botha", 5000, "IT");
      Employee e2 = new Employee("Myra", "Singh", 7600, "SALES");
      Employee e3 = new Employee("Myra", "Singh", 7600, "SALES");

      System.out.println("Getting Employee string representations:");
      System.out.printf("e1: %s%ne2: %s%ne3: %s%n%n", e1, e2, e3);
```

```
Getting Employee string representations:
e1: Employee[first=Prince, last=Botha, salary=5000.0, department=IT]
e2: Employee[first=Myra, last=Singh, salary=7600.0, department=SALES]
e3: Employee[first=Myra, last=Singh, salary=7600.0, department=SALES]
```

```
        System.out.println("Getting data from e1:");
        System.out.printf("first: %s%n", e1.first());
        System.out.printf("last: %s%n", e1.last());
        System.out.printf("salary: %.2f%n", e1.salary());
        System.out.printf("department: %s%n%n", e1.department());

```

```
Getting data from e1:
first: Prince
last: Botha
salary: 5000.00
department: IT
```

```
        System.out.println("Comparing Employee objects for equality:");
        System.out.printf("e1.equals(e2): %b%n", e1.equals(e2));
        System.out.printf("e2.equals(e3): %b%n%n", e2.equals(e3));

```

```
Comparing Employee objects for equality:
e1.equals(e2): false
e2.equals(e3): true
```

```
        System.out.println("Creating Employee with invalid salary:");
        Employee e4 = new Employee("Isabella", "Garcia", -1000, "IT");
    }
}
```

```
Creating Employee with invalid salary:
Exception in thread "main" java.lang.IllegalArgumentException: salary must be
>= 0.0
        at Employee.<init>(Records.java:12)
        at Records.main(Records.java:37)
```

```
// Fig. 8.26: RecordPatternMatchingDemo.java
// Assigning result of a switch expression to a variable.

public class RecordPatternMatchingDemo {
   // record classes representing data for a Rectangle and a Circle
   record Rectangle(double length, double width) { }
   record Circle(double radius) { }

   public static void main(String[] args) {
      Rectangle r = new Rectangle(10, 5);
      Circle c = new Circle(10);
      System.out.printf("Rectangle r: %s%nperimeter: %.2f%n%n",
         r, getPerimeter(r));
      System.out.printf("Circle c : %s%nperimeter: %.2f%n",
         c, getPerimeter(c));
   }

   // uses pattern-matching switch to calculate perimeter of a
   // Rectangle or Circle; throws IllegalArgumentException for other types
   public static double getPerimeter(Object shape) {
      return switch (shape) {
         case Rectangle r -> 2 * r.length() + 2 * r.width();
         case Circle c -> 2 * Math.PI * c.radius() ;
         default -> throw new IllegalArgumentException("invalid type");
      };
   }
}
```

```
Rectangle r: Rectangle[length=10.0, width=5.0]
perimeter: 30.00

Circle c : Circle[radius=10.0]
perimeter: 62.83
```

```java
// Fig. 8.27: RecordPatternMatchingDemo2.java
// Using record patterns in a switch expression to assign
// record members to variables for use in a given case.

public class RecordPatternMatchingDemo2 {
   // record classes representing data for a Rectangle and a Circle
   record Rectangle(double length, double width) { }
   record Circle(double radius) { }

   public static void main(String[] args) {
      Rectangle r = new Rectangle(10, 5);
      Circle c = new Circle(10);
      System.out.printf("Rectangle r: %s%nperimeter: %.2f%n%n",
         r, getPerimeter(r));
      System.out.printf("Circle c : %s%nperimeter: %.2f%n",
         c, getPerimeter(c));
   }

   // uses pattern-matching switch to calculate perimeter of a
   // Rectangle or Circle; throws IllegalArgumentException for other types
   public static double getPerimeter(Object shape) {
      return switch (shape) {
         case Rectangle(var length, var width) -> 2 * length + 2 * width;
         case Circle(var radius) -> 2 * Math.PI * radius;
         default -> throw new IllegalArgumentException("invalid type");
      };
   }
}
```

```
Rectangle r: Rectangle[length=10.0, width=5.0]
perimeter: 30.00

Circle c : Circle[radius=10.0]
perimeter: 62.83
```

```java
// Fig. 9.1: SalariedEmployee.java
// SalariedEmployee class represents an employee paid a fixed salary.
import java.math.BigDecimal;

public class SalariedEmployee extends Object {
   private final String name;
   private BigDecimal salary; // fixed weekly Salary

   // two-argument constructor
   public SalariedEmployee(String name, BigDecimal salary) {
      // implicit call to Object's default constructor occurs here

      // if salary is invalid throw exception
      if (salary.compareTo(BigDecimal.ZERO) < 0) {
         throw new IllegalArgumentException("salary must be >= 0.0");
      }

      this.name = name;
      this.salary = salary;
   }

   // return name
   public String getName() {return name;}

   // set salary amount
   public void setSalary(BigDecimal salary) {
      if (salary.compareTo(BigDecimal.ZERO) < 0) {
         throw new IllegalArgumentException("salary must be >= 0.0");
      }

      this.salary = salary;
   }

   // return salary amount
   public BigDecimal getSalary() {return salary;}

   // calculate earnings
   public BigDecimal calculateEarnings() {return getSalary();}

   // return String representation of SalariedEmployee object
   @Override // indicates that this method overrides a superclass method
   public String toString() {
      return String.format("name: %s%nsalary: $%s",
         getName(), getSalary());
   }
}
```

```java
// Fig. 9.2: SalariedEmployeeTest.java
// SalariedEmployee class test program.
import java.math.BigDecimal;
import java.math.RoundingMode;

public class SalariedEmployeeTest {
   public static void main(String[] args) {
      // instantiate SalariedEmployee object
      var employee = new SalariedEmployee(
         "Sierra Dembo", new BigDecimal("1000.00"));

      // get SalariedEmployee data
      System.out.println("Employee information obtained by get methods:");
      System.out.printf("name: %s%nsalary: $%s%n", employee.getName(),
         employee.getSalary().setScale(2, RoundingMode.HALF_EVEN));

      employee.setSalary(new BigDecimal("1200.00")); // change salary

      // display updated information and calculate and display earnings
      System.out.printf("%n%s:%n%s%n%nearnings: $%s%n",
         "Updated employee information obtained by toString", employee,
         employee.calculateEarnings().setScale(
            2, RoundingMode.HALF_EVEN));
   }
}
```

```
Employee information obtained by get methods:
name: Sierra Dembo
salary: $1000.00

Updated employee information obtained by toString:
name: Sierra Dembo
salary: $1200.00

earnings: $1200.00
```

```java
// Fig. 9.3: SalariedCommissionEmployee.java
// SalariedCommissionEmployee class inherits from SalariedEmployee and
// accesses the superclass's private data via inherited public methods.
import java.math.BigDecimal;

public class SalariedCommissionEmployee extends SalariedEmployee {
   private BigDecimal grossSales; // gross weekly sales
   private BigDecimal commissionRate; // commission percentage

   // four-argument constructor
   public SalariedCommissionEmployee(String name, BigDecimal salary,
      BigDecimal grossSales, BigDecimal commissionRate) {

      super(name, salary); // call superclass constructor

      // if grossSales is invalid throw exception
      if (grossSales.compareTo(BigDecimal.ZERO) < 0) {
         throw new IllegalArgumentException("gross sales must be >= 0.0");
      }

      // if commissionRate is invalid throw exception
      if (commissionRate.compareTo(BigDecimal.ZERO) <= 0 ||
          commissionRate.compareTo(BigDecimal.ONE) >= 0) {
         throw new IllegalArgumentException(
            "commission rate must be > 0.0 and < 1.0");
      }

      this.grossSales = grossSales;
      this.commissionRate = commissionRate;
   }

   // set gross sales
   public void setGrossSales(BigDecimal grossSales) {
      if (grossSales.compareTo(BigDecimal.ZERO) < 0) {
         throw new IllegalArgumentException("gross sales must be >= 0.0");
      }

```

```
38          this.grossSales = grossSales;
39       }
40
41       // return gross sales
42       public BigDecimal getGrossSales() {return grossSales;}
43
44       // set commission rate
45       public void setCommissionRate(BigDecimal commissionRate) {
46          if (commissionRate.compareTo(BigDecimal.ZERO) <= 0 ||
47              commissionRate.compareTo(BigDecimal.ONE) >= 0) {
48             throw new IllegalArgumentException(
49                "commission rate must be > 0.0 and < 1.0");
50          }
51
52          this.commissionRate = commissionRate;
53       }
54
55       // return commission rate
56       public BigDecimal getCommissionRate() {return commissionRate;}
57
58       // calculate earnings
59       @Override
60       public BigDecimal calculateEarnings() {
61          return super.calculateEarnings().add(
62             getGrossSales().multiply(getCommissionRate()));
63       }
64
65       // return String representation of SalariedCommissionEmployee
66       @Override
67       public String toString() {
68          return String.format("%s%ngross sales: $%s%ncommission rate: %s",
69             super.toString(), getGrossSales(), getCommissionRate());
70       }
71    }
```

```
// Fig. 9.4: SalariedCommissionEmployeeTest.java
// SalariedCommissionEmployee test program.
import java.math.BigDecimal;
import java.math.RoundingMode;

public class SalariedCommissionEmployeeTest {
   public static void main(String[] args) {
      // instantiate SalariedCommissionEmployee object
      var employee = new SalariedCommissionEmployee("James Davis",
         new BigDecimal("500.00"), new BigDecimal("10000.00"),
         new BigDecimal("0.04"));

      // get SalariedCommissionEmployee data
      System.out.printf("""
         Employee information obtained by get methods:
         name: %s
         salary: $%s
         gross sales: $%s
         commission rate: %s
         """, employee.getName(), employee.getSalary(),
         employee.getGrossSales(), employee.getCommissionRate());

      // update gross sales and commission rate
      employee.setGrossSales(new BigDecimal("8000.00"));
      employee.setCommissionRate(new BigDecimal("0.1"));

      // display updated information and calculate and display earnings
      System.out.printf("%n%s:%n%s%n%nearnings: $%s%n",
         "Updated employee information obtained by toString", employee,
         employee.calculateEarnings().setScale(
            2, RoundingMode.HALF_EVEN));
   }
}
```

```
Employee information obtained by get methods:
name: James Davis
salary: $500.00
gross sales: $10000.00
commission rate: 0.04

Updated employee information obtained by toString:
name: James Davis
salary: $500.00
gross sales: $8000.00
commission rate: 0.1

earnings: $1300.00
```

```
// Fig. 9.5: PolymorphismTest.java
// Assigning superclass and subclass references to superclass and
// subclass variables.
import java.math.BigDecimal;
import java.math.RoundingMode;

public class PolymorphismTest {
   public static void main(String[] args) {
      // assign superclass reference to superclass variable
      var salariedEmployee = new SalariedEmployee("Sierra Dembo",
         new BigDecimal("1000.00"));

      // assign subclass reference to subclass variable
      var salariedCommissionEmployee = new SalariedCommissionEmployee(
         "James Davis", new BigDecimal("500.00"),
         new BigDecimal("10000.00"), new BigDecimal("0.04"));

      // invoke toString on superclass object using superclass variable
      System.out.printf("""
         Call SalariedEmployee's toString and calculateEarnings
         methods with superclass reference to superclass object:
         %s
         earnings: $%s%n%n""", salariedEmployee.toString(),
         salariedEmployee.calculateEarnings().setScale(
            2, RoundingMode.HALF_EVEN));

      // invoke toString on subclass object using subclass variable
      System.out.printf("""
         Call SalariedCommissionEmployee's toString and calculateEarnings
         methods with subclass reference to subclass object:
         %s
         earnings: $%s%n%n""", salariedCommissionEmployee.toString(),
```

```
            salariedCommissionEmployee.calculateEarnings().setScale(
               2, RoundingMode.HALF_EVEN));

         // invoke toString on subclass object using superclass variable
         SalariedEmployee salariedEmployee2 = salariedCommissionEmployee;
         System.out.printf("""
            Call SalariedCommissionEmployee's toString and calculateEarnings
            methods with superclass reference to subclass object:
            %s
            earnings: $%s%n""", salariedEmployee2.toString(),
            salariedEmployee2.calculateEarnings().setScale(
               2, RoundingMode.HALF_EVEN));
      }
   }
```

```
Call SalariedEmployee's toString and calculateEarnings
methods with superclass reference to superclass object:
name: Sierra Dembo
salary: $1000.00
earnings: $1000.00

Call SalariedCommissionEmployee's toString and calculateEarnings
methods with subclass reference to subclass object:
name: James Davis
salary: $500.00
gross sales: $10000.00
commission rate: 0.04
earnings: $900.00

Call SalariedCommissionEmployee's toString and calculateEarnings
methods with superclass reference to subclass object:
name: James Davis
salary: $500.00
gross sales: $10000.00
commission rate: 0.04
earnings: $900.00
```

```
public abstract void draw(); // abstract method
```

```
// Fig. 9.6: Employee.java
// Employee abstract superclass.
import java.math.BigDecimal;

public abstract class Employee {
   private final String name;

   // constructor
   public Employee(String name) {this.name = name;}

   // return name
   public String getName() {return name;}

   // return String representation of Employee object
   @Override
   public String toString() {return String.format("name: %s", getName());}

   // abstract method must be overridden by concrete subclasses
   public abstract BigDecimal calculateEarnings(); // no implementation
}
```

```java
// Fig. 9.7: SalariedEmployee.java
// SalariedEmployee concrete class extends abstract class Employee.
import java.math.BigDecimal;

public class SalariedEmployee extends Employee {
   private BigDecimal salary; // fixed weekly Salary

   // constructor
   public SalariedEmployee(String name, BigDecimal salary) {
      super(name);

      // if salary is invalid throw exception
      if (salary.compareTo(BigDecimal.ZERO) < 0) {
         throw new IllegalArgumentException("salary must be >= 0.0");
      }

      this.salary = salary;
   }

   // set salary amount
   public void setSalary(BigDecimal salary) {
      if (salary.compareTo(BigDecimal.ZERO) < 0) {
         throw new IllegalArgumentException("salary must be >= 0.0");
      }

      this.salary = salary;
   }

   // return salary amount
   public BigDecimal getSalary() {return salary;}

   // calculate earnings
   @Override
   public BigDecimal calculateEarnings() {return getSalary();}

   // return String representation of SalariedEmployee object
   @Override // indicates that this method overrides a superclass method
   public String toString() {
      return String.format("%s%nsalary: $%s",
         super.toString(), getSalary());
   }
}
```

```java
// Fig. 9.8: CommissionEmployee.java
// CommissionEmployee class extends Employee.
import java.math.BigDecimal;

public class CommissionEmployee extends Employee {
   private BigDecimal grossSales; // gross weekly sales
   private BigDecimal commissionRate; // commission percentage

   // constructor
   public CommissionEmployee(String name,
      BigDecimal grossSales, BigDecimal commissionRate) {
      super(name);

      // if grossSales is invalid throw exception
      if (grossSales.compareTo(BigDecimal.ZERO) < 0) {
         throw new IllegalArgumentException("gross sales must be >= 0.0");
      }

      // if commissionRate is invalid throw exception
      if (commissionRate.compareTo(BigDecimal.ZERO) <= 0 ||
          commissionRate.compareTo(BigDecimal.ONE) >= 0) {
         throw new IllegalArgumentException(
            "commission rate must be > 0.0 and < 1.0");
      }

      this.grossSales = grossSales;
      this.commissionRate = commissionRate;
   }

```

```
30      // set gross sales
31      public void setGrossSales(BigDecimal grossSales) {
32         if (grossSales.compareTo(BigDecimal.ZERO) < 0) {
33            throw new IllegalArgumentException("gross sales must be >= 0.0");
34         }
35
36         this.grossSales = grossSales;
37      }
38
39      // return gross sales
40      public BigDecimal getGrossSales() {return grossSales;}
41
42      // set commission rate
43      public void setCommissionRate(BigDecimal commissionRate) {
44         if (commissionRate.compareTo(BigDecimal.ZERO) <= 0 ||
45             commissionRate.compareTo(BigDecimal.ONE) >= 0) {
46            throw new IllegalArgumentException(
47               "commission rate must be > 0.0 and < 1.0");
48         }
49
50         this.commissionRate = commissionRate;
51      }
52
53      // return commission rate
54      public BigDecimal getCommissionRate() {return commissionRate;}
55
56      // calculate earnings
57      @Override
58      public BigDecimal calculateEarnings() {
59         return getGrossSales().multiply(getCommissionRate());
60      }
61
62      // return String representation of CommissionEmployee
63      @Override
64      public String toString() {
65         return String.format("%s%ngross sales: $%s%ncommission rate: %s",
66            super.toString(), getGrossSales(), getCommissionRate());
67      }
68   }
```

```
1   // Fig. 9.9: PayrollSystemTest.java
2   // Employee hierarchy test program.
3   import java.math.BigDecimal;
4   import java.math.RoundingMode;
5
6   public class PayrollSystemTest {
7      public static void main(String[] args) {
8         // create subclass objects
9         var salariedEmployee = new SalariedEmployee("Sierra Dembo",
10           new BigDecimal("1000.00"));
11        var commissionEmployee = new CommissionEmployee("James Davis",
12           new BigDecimal("10000.00"), new BigDecimal("0.06"));
13
14        System.out.println("Employees processed individually:");
15
16        System.out.printf("%s%nearned: $%s%n%n", salariedEmployee,
17           salariedEmployee.calculateEarnings().setScale(
18              2, RoundingMode.HALF_EVEN));
19        System.out.printf("%s%nearned: $%s%n%n", commissionEmployee,
20           commissionEmployee.calculateEarnings().setScale(
21              2, RoundingMode.HALF_EVEN));
22
23        // create and initialize Employee array
24        Employee[] employees = {salariedEmployee, commissionEmployee};
25
26        System.out.println("Employees processed polymorphically:");
27
28        // generically process each element in array employees
29        for (Employee currentEmployee : employees) {
30           System.out.println(currentEmployee); // invokes toString
31
32           // determine whether element is a SalariedEmployee
33           if (currentEmployee instanceof SalariedEmployee employee) {
34              employee.setSalary(
35                 employee.getSalary().multiply(new BigDecimal("1.10")));
```

```
36
37                System.out.printf(
38                   "new salary with 10%% increase is: $%s%n",
39                   employee.getSalary().setScale(2, RoundingMode.HALF_EVEN));
40             }
41
42             System.out.printf("earned: $%s%n%n",
43                currentEmployee.calculateEarnings().setScale(
44                   2, RoundingMode.HALF_EVEN));
45          }
46
47          // get type name of each object in employees array
48          for (int j = 0; j < employees.length; ++j) {
49             System.out.printf("Employee %d is a %s%n", j,
50                employees[j].getClass().getName());
51          }
52       }
53    }
```

```
Employees processed individually:
name: Sierra Dembo
salary: $1000.00
earned: $1000.00

name: James Davis
gross sales: $10000.00
commission rate: 0.06
earned: $600.00

Employees processed polymorphically:
name: Sierra Dembo
salary: $1000.00
new salary with 10% increase is: $1100.00
earned: $1100.00

name: James Davis
gross sales: $10000.00
commission rate: 0.06
earned: $600.00

Employee 0 is a SalariedEmployee
Employee 1 is a CommissionEmployee
```

```
if (currentEmployee instanceof SalariedEmployee) {
   // downcast Employee reference to SalariedEmployee reference
   SalariedEmployee employee = (SalariedEmployee) currentEmployee;

   // rest of if statement body
}
```

https://docs.oracle.com/javase/tutorial/java/IandI/final.html

`public class` *ClassName* `extends` *SuperclassName* `implements` *InterfaceName*

`public class` *ClassName* `implements` *InterfaceName*

```
1  // Fig. 9.10: Payable.java
2  // Payable interface declaration.
3  import java.math.BigDecimal;
4
5  public interface Payable {
6     public abstract BigDecimal calculatePayment(); // no implementation
7  }
```

```java
// Fig. 9.11: Invoice.java
// Invoice class that implements Payable.
import java.math.BigDecimal;

public class Invoice implements Payable {
   private final String partNumber;
   private final String partDescription;
   private final int quantity;
   private final BigDecimal pricePerItem;

   // constructor
   public Invoice(String partNumber, String partDescription, int quantity,
      BigDecimal pricePerItem) {
      if (quantity < 0) { // validate quantity
         throw new IllegalArgumentException("Quantity must be >= 0");
      }

      // validate pricePerItem
      if (pricePerItem.compareTo(BigDecimal.ZERO) < 0) {
         throw new IllegalArgumentException(
            "Price per item must be >= 0");
      }

      this.quantity = quantity;
      this.partNumber = partNumber;

      this.partDescription = partDescription;
      this.pricePerItem = pricePerItem;
   }

   // get methods
   public String getPartNumber() {return partNumber;}
   public String getPartDescription() {return partDescription;}
   public int getQuantity() {return quantity;}
   public BigDecimal getPricePerItem() {return pricePerItem;}

   // return String representation of Invoice object
   @Override
   public String toString() {
      return String.format("%s: %s (%s)%n%s: %d%n%s: $%s",
         "part number", getPartNumber(), getPartDescription(),
         "quantity", getQuantity(), "price per item", getPricePerItem());
   }

   // method required to carry out contract with interface Payable
   @Override
   public BigDecimal calculatePayment() {
      BigDecimal quantity = new BigDecimal(getQuantity());
      return quantity.multiply(getPricePerItem()); // calculate total cost
   }
}
```

```java
// Fig. 9.12: Employee.java
// Employee abstract superclass that implements interface Payable.
import java.math.BigDecimal;

public abstract class Employee implements Payable {
   private final String name;

   // constructor
   public Employee(String name) {this.name = name;}

   // return name
   public String getName() {return name;}

   // return String representation of Employee object
   @Override
   public String toString() {return String.format("name: %s", getName());}

   // abstract method must be overridden by concrete subclasses
   public abstract BigDecimal calculateEarnings(); // no implementation

   // implementing calculatePayment here enables the entire Employee
   // class hierarchy to be used in an app that processes Payables
   @Override
   public BigDecimal calculatePayment() {return calculateEarnings();}
}
```

```
1  // Fig. 9.13: PayableInterfaceTest.java
2  // Payable interface test program processing Invoices and
3  // Employees polymorphically.
4  import java.math.BigDecimal;
5  import java.math.RoundingMode;
6
7  public class PayableInterfaceTest {
8     public static void main(String[] args) {
9        // create four-element Payable array
10       Payable[] payableObjects = {
11          new Invoice("01234", "seat", 2, new BigDecimal("375.00")),
12          new Invoice("56789", "tire", 4, new BigDecimal("79.95")),
13          new SalariedEmployee("Lorena Silva", new BigDecimal("1200.00")),
14          new SalariedEmployee("Hans Olsen", new BigDecimal("800.00"))
15       };
16
17       System.out.println(
18          "Invoices and Employees processed polymorphically:\n");
19
20       // generically process each element in array payableObjects
21       for (Payable currentPayable : payableObjects) {
22          // output currentPayable and its appropriate payment amount
23          System.printf("%s:%n%s %npayment due: $%s%n%n",
24             currentPayable.getClass().getName(),
25             currentPayable.toString(), // could invoke implicitly

26             currentPayable.calculatePayment().setScale(
27                2, RoundingMode.HALF_EVEN));
28       }
29    }
30 }
```

```
Invoices and Employees processed polymorphically:

Invoice:
part number: 01234 (seat)
quantity: 2
price per item: $375.00
payment due: $750.00
```

```
Invoice:
part number: 56789 (tire)
quantity: 4
price per item: $79.95
payment due: $319.80

SalariedEmployee:
name: Lorena Silva
salary: $1200.00
payment due: $1200.00

SalariedEmployee:
name: Hans Olsen
salary: $800.00
payment due: $800.00
```

```
public default returnType methodName(parameters) {
   statement(s)
}
```

```
1  // Fig. 9.14: CompensationModel.java
2  // CompensationModel interface defining method calculateEarnings.
3  import java.math.BigDecimal;
4
5  public interface CompensationModel {
6     public abstract BigDecimal calculateEarnings();
7  }
```

```java
// Fig. 9.15: Employee.java
// An Employee "has a" CompensationModel.
import java.math.BigDecimal;

public final class Employee {
   private final String name;
   private CompensationModel model; // has-a relationship

   // uses "constructor injection" to initialize the model
   // with a reference to a CompensationModel implementation
   public Employee(String name, CompensationModel model) {
      this.name = name;
      this.model = model; // constructor injection
   }

   // set method performs "property injection" to change the
   // model to a new CompensationModel implementation
   public void setCompensationModel(CompensationModel model) {
      this.model = model; // property injection
   }

   // return the current CompensationModel
   public CompensationModel getCompensationModel() {return model;}

   // use the CompensationModel to calculate the Employee's earnings
   public BigDecimal calculateEarnings() {
      return model.calculateEarnings();
   }

   // return String representation of Employee object
   @Override
   public String toString() {
      return String.format("%s\n%s", name, model.toString());
   }
}
```

```java
// Fig. 9.16: Salaried.java
// Salaried implements the CompensationModel interface.
import java.math.BigDecimal;

public final class Salaried implements CompensationModel {
   private BigDecimal salary;

   // constructor
   public Salaried(BigDecimal salary) {
      // if salary is invalid throw exception
      if (salary.compareTo(BigDecimal.ZERO) < 0) {
         throw new IllegalArgumentException("salary must be >= 0.0");
      }

      this.salary = salary;
   }

   // return the salary
   public BigDecimal getSalary() {return salary;}

   // override CompensationModel abstract method calculateEarnings
   @Override
   public BigDecimal calculateEarnings() {return salary;}

   // override Object method toString
   @Override
   public String toString() {
      return String.format("salary: $%s", salary);
   }
}
```

```java
// Fig. 9.17: Commission.java
// Commission implements the CompensationModel interface.
import java.math.BigDecimal;

public final class Commission implements CompensationModel {
   private BigDecimal grossSales;
   private BigDecimal commissionRate;

   // constructor
   public Commission(BigDecimal grossSales, BigDecimal commissionRate) {
      // if grossSales is invalid throw exception
      if (grossSales.compareTo(BigDecimal.ZERO) < 0) {
         throw new IllegalArgumentException("gross sales must be >= 0.0");
      }

      // if commissionRate is invalid throw exception
      if (commissionRate.compareTo(BigDecimal.ZERO) <= 0 ||
          commissionRate.compareTo(BigDecimal.ONE) >= 0) {
         throw new IllegalArgumentException(
            "commission rate must be > 0.0 and < 1.0");
      }


      this.grossSales = grossSales;
      this.commissionRate = commissionRate;
   }

   // return the grossSales
   public BigDecimal getGrossSales() {return grossSales;}

   // return the commissionRate
   public BigDecimal getCommissionRate() {return commissionRate;}

   // override CompensationModel abstract method calculateEarnings
   @Override
   public BigDecimal calculateEarnings() {
      return grossSales.multiply(commissionRate);
   }

   // override Object method toString
   @Override
   public String toString() {
      return String.format("gross sales: $%s; commission rate: %s",
         grossSales, commissionRate);
   }
}
```

```java
// Fig. 9.18: PayrollSystemTest.java
// Processing Employees with various CompensationModels.
import java.math.BigDecimal;
import java.math.RoundingMode;

public class PayrollSystemTest {
   public static void main(String[] args) {
      // create CompensationModels and Employees
      var salaried = new Salaried(new BigDecimal("1000.00"));
      var salariedEmployee = new Employee("Sierra Dembo", salaried);

      var commission = new Commission(
         new BigDecimal("10000.00"), new BigDecimal("0.06"));
      var commissionEmployee = new Employee("James Davis", commission);

      // create and initialize array of Employees
      Employee[] employees = {salariedEmployee, commissionEmployee};

```

```
        // print each Employee's information and earnings
        for (Employee employee : employees) {
           // if Salaried CompensationModel, increase salary by 10%
           if (employee.getCompensationModel() instanceof Salaried s) {
              BigDecimal newSalary =
                 s.getSalary().multiply(new BigDecimal("1.10"));

              // inject new CompensationModel
              employee.setCompensationModel(new Salaried(
                 newSalary.setScale(2, RoundingMode.HALF_EVEN)));
           }

           System.out.printf("%s%nearned: $%s%n%n", employee,
              employee.calculateEarnings().setScale(
                 2, RoundingMode.HALF_EVEN));
        }
     }
  }
```

```
Sierra Dembo
salary: $1100.00
earned: $1100.00

James Davis
gross sales: $10000.00; commission rate: 0.06
earned: $600.00
```

```
// Fig. 9.19: CompensationModel.java
// sealed CompensationModel interface defining method calculateEarnings.
import java.math.BigDecimal;

public sealed interface CompensationModel permits Salaried, Commission {
   public abstract BigDecimal calculateEarnings();
}
```

```
public sealed interface NewInterface extends SuperInterface
   permits ListOfImplementingClasses {
   // interface members
}

public sealed class NewClass extends Superclass implements ListOfInterfaces
   permits ListOfSubclasses {
   // class members
}
```

```
// Fig. 9.20: PayrollSystemTest.java
// Processing Employees with various CompensationModels.
import java.math.BigDecimal;
import java.math.RoundingMode;

public class PayrollSystemTest {
   public static void main(String[] args) {
      // create CompensationModels and Employees
      var salaried = new Salaried(new BigDecimal("1000.00"));
      var salariedEmployee = new Employee("Sierra Dembo", salaried);

      var commission = new Commission(
         new BigDecimal("10000.00"), new BigDecimal("0.06"));
      var commissionEmployee = new Employee("James Davis", commission);

      // create and initialize array of Employees
      Employee[] employees = {salariedEmployee, commissionEmployee};

      // print each Employee's information and earnings
      for (Employee employee : employees) {
         // if Salaried CompensationModel, increase salary by 10%
         BigDecimal earnings =
            switch (employee.getCompensationModel()) {
               case Salaried s -> {
                  BigDecimal newSalary =
                     s.getSalary().multiply(new BigDecimal("1.10"));

                  // inject new CompensationModel
                  employee.setCompensationModel(new Salaried(
                     newSalary.setScale(2, RoundingMode.HALF_EVEN)));
                  yield employee.calculateEarnings();
               }
               case Commission c -> c.calculateEarnings();
            };

         System.out.printf("%s%nearned: $%s%n%n",
            employee, earnings.setScale(2, RoundingMode.HALF_EVEN));
      }
   }
}
```

```
Sierra Dembo
salary: $1100.00
earned: $1100.00

James Davis
gross sales: $10000.00; commission rate: 0.06
earned: $600.00
```

```java
// Fig. 10.1: DivideByZeroNoExceptionHandling.java
// Integer division without exception handling.
import java.util.Scanner;

public class DivideByZeroNoExceptionHandling {
   // demonstrates throwing an exception when a divide-by-zero occurs
   public static int quotient(int numerator, int denominator) {
      return numerator / denominator; // possible division by zero
   }

   public static void main(String[] args) {
      var input = new Scanner(System.in);

      System.out.print("Enter an integer numerator: ");
      int numerator = input.nextInt();
      System.out.print("Enter an integer denominator: ");
      int denominator = input.nextInt();

      int result = quotient(numerator, denominator);
      System.out.printf(
         "%nResult: %d / %d = %d%n", numerator, denominator, result);
   }
}
```

```
Enter an integer numerator: 100
Enter an integer denominator: 7

Result: 100 / 7 = 14
```

```
Enter an integer numerator: 100
Enter an integer denominator: 0
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at DivideByZeroNoExceptionHandling.quotient(
           DivideByZeroNoExceptionHandling.java:8)
        at DivideByZeroNoExceptionHandling.main(
           DivideByZeroNoExceptionHandling.java:19)
```

```
Enter an integer numerator: hello
Exception in thread "main" java.util.InputMismatchException
        at java.base/java.util.Scanner.throwFor(Scanner.java:964)
        at java.base/java.util.Scanner.next(Scanner.java:1619)
        at java.base/java.util.Scanner.nextInt(Scanner.java:2284)
        at java.base/java.util.Scanner.nextInt(Scanner.java:2238)
        at DivideByZeroNoExceptionHandling.main(
           DivideByZeroNoExceptionHandling.java:15)
```

```java
// Fig. 10.2: DivideByZeroWithExceptionHandling.java
// Handling ArithmeticExceptions and InputMismatchExceptions.
import java.util.InputMismatchException;
import java.util.Scanner;

public class DivideByZeroWithExceptionHandling {
   // demonstrates throwing an exception when a divide-by-zero occurs
   public static int quotient(int numerator, int denominator)
      throws ArithmeticException {
      return numerator / denominator; // possible division by zero
   }

   public static void main(String[] args) {
      var input = new Scanner(System.in);
      boolean continueLoop = true; // determines if more input is needed

      while (continueLoop) {
         try { // read two numbers and calculate quotient
            System.out.print("Enter an integer numerator: ");
            int numerator = input.nextInt();
            System.out.print("Enter an integer denominator: ");
            int denominator = input.nextInt();
```

```
23
24              int result = quotient(numerator, denominator);
25              System.out.printf("%nResult: %d / %d = %d%n", numerator,
26                 denominator, result);
27              continueLoop = false; // input successful; end looping
28           }
29           catch (InputMismatchException inputMismatchException) {
30              System.err.printf("%nException: %s%n",
31                 inputMismatchException);
32              input.nextLine(); // discard input so user can try again
33              System.out.printf(
34                 "You must enter integers. Try again.%n%n");
35           }
36           catch (ArithmeticException arithmeticException) {
37              System.err.printf("%nException: %s%n", arithmeticException);
38              System.out.printf(
39                 "Zero is an invalid denominator. Try again.%n%n");
40           }
41        }
42     }
43  }
```

```
Enter an integer numerator: hello

Exception: java.util.InputMismatchException
You must enter integers. Try again.

Enter an integer numerator: 100
Enter an integer denominator: 0

Exception: java.lang.ArithmeticException: / by zero
Zero is an invalid denominator. Try again.

Enter an integer numerator: 100
Enter an integer denominator: 7

Result: 100 / 7 = 14
```

catch (*Type1* | *Type2* | *Type3* e)

https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/lang/Throwable.html

```
1  // Fig. 10.3: UsingExceptions.java
2  // try...catch...finally exception handling mechanism.
3
4  public class UsingExceptions {
5     public static void main(String[] args) {
6        try {
7           throwException();
8        }
9        catch (Exception exception) { // exception thrown by throwException
10          System.err.println("Exception handled in main");
11       }
12
13       doesNotThrowException();
14    }
15
16    // demonstrate try...catch...finally
17    public static void throwException() throws Exception {
18       try { // throw an exception and immediately catch it
19          System.out.println("Method throwException");
20          throw new Exception(); // generate exception
21       }
22       catch (Exception exception) { // catch exception thrown in try
23          System.err.println(
24             "Exception handled in method throwException");
25          throw exception; // rethrow for further processing
26
27          // code here would not be reached; would cause compilation errors
28
```

```
        }
        finally { // executes regardless of what occurs in try...catch
           System.err.println("Finally executed in throwException");
        }

        // code here would not be reached; would cause compilation errors
     }

     // demonstrate finally when no exception occurs
     public static void doesNotThrowException() {
        try { // try block does not throw an exception
           System.out.println("Method doesNotThrowException");
        }
        catch (Exception exception) { // does not execute
           System.err.println(exception);
        }
        finally { // executes regardless of what occurs in try...catch
           System.err.println("Finally executed in doesNotThrowException");
        }

        System.out.println("End of method doesNotThrowException");
     }
  }
```

```
Method throwException
Exception handled in method throwException
Finally executed in throwException
Exception handled in main
Method doesNotThrowException
Finally executed in doesNotThrowException
End of method doesNotThrowException
```

```
  // Fig. 10.4: UsingExceptions.java
  // Stack unwinding and obtaining data from an exception object.

  public class UsingExceptions {
     public static void main(String[] args) {
        try {
           method1();
        }
        catch (Exception exception) { // catch exception thrown in method1
           System.err.printf("%s%n%n", exception.getMessage());
           exception.printStackTrace();

           // obtain the stack-trace information
           StackTraceElement[] traceElements = exception.getStackTrace();

           System.out.printf("%nStack trace from getStackTrace:%n");
           System.out.println("Class\t\tFile\t\t\tLine\tMethod");

           // loop through traceElements to get exception description
           for (StackTraceElement element : traceElements) {
              System.out.printf("%s\t", element.getClassName());
              System.out.printf("%s\t", element.getFileName());
              System.out.printf("%s\t", element.getLineNumber());
              System.out.printf("%s%n", element.getMethodName());
           }
        }
     }

     // call method2; throw exceptions back to main
```

```
30      public static void method1() throws Exception {
31         method2();
32      }
33
34      // call method3; throw exceptions back to method1
35      public static void method2() throws Exception {
36         method3();
37      }
38
39      // throw Exception back to method2
40      public static void method3() throws Exception {
41         throw new Exception("Exception thrown in method3");
42      }
43   }
```

```
Exception thrown in method3

java.lang.Exception: Exception thrown in method3
        at UsingExceptions.method3(UsingExceptions.java:41)
        at UsingExceptions.method2(UsingExceptions.java:36)
        at UsingExceptions.method1(UsingExceptions.java:31)
        at UsingExceptions.main(UsingExceptions.java:7)

Stack trace from getStackTrace:
Class           File                    Line    Method
UsingExceptions UsingExceptions.java    41      method3
UsingExceptions UsingExceptions.java    36      method2
UsingExceptions UsingExceptions.java    31      method1
UsingExceptions UsingExceptions.java    7       main
```

className.*methodName*(*fileName*:*lineNumber*)

```
1   // Fig. 10.5: UsingChainedExceptions.java
2   // Chained exceptions.
3
4   public class UsingChainedExceptions {
5      public static void main(String[] args) {
6         try {
7            method1();
8         }
9         catch (Exception exception) { // exceptions thrown from method1
10           exception.printStackTrace();
11        }
12     }
13
14     // call method2; throw exceptions back to main
15     public static void method1() throws Exception {
16        try {
17           method2();
18        }
19        catch (Exception exception) { // exception thrown from method2
20           throw new Exception("Exception thrown in method1", exception);
21        }
22     }
23
24     // call method3; throw exceptions back to method1
25     public static void method2() throws Exception {
26        try {
```

```
            method3();
         }
         catch (Exception exception) { // exception thrown from method3
            throw new Exception("Exception thrown in method2", exception);
         }
      }

      // throw Exception back to method2
      public static void method3() throws Exception {
         throw new Exception("Exception thrown in method3");
      }
   }
```

```
java.lang.Exception: Exception thrown in method1
        at UsingChainedExceptions.method1(UsingChainedExceptions.java:20)
        at UsingChainedExceptions.main(UsingChainedExceptions.java:7)
Caused by: java.lang.Exception: Exception thrown in method2
        at UsingChainedExceptions.method2(UsingChainedExceptions.java:30)
        at UsingChainedExceptions.method1(UsingChainedExceptions.java:17)
        ... 1 more
Caused by: java.lang.Exception: Exception thrown in method3
        at UsingChainedExceptions.method3(UsingChainedExceptions.java:36)
        at UsingChainedExceptions.method2(UsingChainedExceptions.java:27)
        ... 2 more
```

```
assert expression1 : expression2;
```

```
// Fig. 10.6: AssertTest.java
// Checking with assert that a value is within range
import java.util.Scanner;

public class AssertTest {
   public static void main(String[] args) {
      var input = new Scanner(System.in);

      System.out.print("Enter a number between 0 and 10: ");
      int number = input.nextInt();

      // assert that the value is >= 0 and <= 10
      assert (number >= 0 && number <= 10) : "bad number: " + number;

      System.out.printf("You entered %d%n", number);
   }
}
```

```
Enter a number between 0 and 10: 5
You entered 5
```

```
Enter a number between 0 and 10: 50
Exception in thread "main" java.lang.AssertionError: bad number: 50
        at AssertTest.main(AssertTest.java:13)
```

```
try (var theObject = new ResourceClass()) {
   // Use theObject here. Resources automatically released when the
   // try block terminates successfully or due to an exception.
}
catch (Exception e) {
   // Catch exceptions that occur while using the resource.
}

var theObject = new ResourceClass();

try (theObject) {
   // Use theObject here. Resources automatically released when the
   // try block terminates successfully or due to an exception.
}
catch (Exception e) {
   // Catch exceptions that occur while using the resource.
}
```

```java
// Fig. 10.7: UnnamedVariableDemo.java
// Using _ where a variable name is required.
public class UnnamedVariableDemo {
   public static void main(String[] args) {
      int[] array = {1, 2, 3, 4, 5};

      try {
         int value = array[10]; // index 10 is out of bounds
      }
      catch (ArrayIndexOutOfBoundsException _) {
         System.out.println("Index out of range");
      }
   }
}
```

```
Index out of range
```

```java
// Fig. 11.1: FileAndDirectoryInfo.java
// File class used to obtain file and directory information.
import java.io.IOException;
import java.nio.file.DirectoryStream;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.Scanner;

public class FileAndDirectoryInfo {
   public static void main(String[] args) throws IOException {
      var input = new Scanner(System.in);

      System.out.println("Enter file or directory name:");

      // create Path object based on user input
      Path path = Path.of(input.nextLine());

      if (Files.exists(path)) { // if path exists, output info about it
         // display file or directory information
         System.out.printf("%n%s exists%n", path.getFileName());
         System.out.printf("%s a directory%n",
            Files.isDirectory(path) ? "Is" : "Is not");
         System.out.printf("%s an absolute path%n",
            path.isAbsolute() ? "Is" : "Is not");

         System.out.printf("Last modified: %s%n",
            Files.getLastModifiedTime(path));
         System.out.printf("Size: %s%n", Files.size(path));
         System.out.printf("Path: %s%n", path);
         System.out.printf("Absolute path: %s%n", path.toAbsolutePath());

         if (Files.isDirectory(path)) { // output directory listing
            System.out.printf("%nDirectory contents:%n");

            // object for iterating through a directory's contents
            DirectoryStream<Path> directoryStreamPath =
               Files.newDirectoryStream(path);

            for (Path currentPath : directoryStreamPath) {
               System.out.println(currentPath);
            }
         }
      }
      else { // not file or directory, output error message
         System.out.printf("%s does not exist%n", path);
      }
   }
}
```

```
Enter file or directory name:
c:\Users\pauldeitel\Documents\examples\ch11

ch11 exists
Is a directory
Is an absolute path
Last modified: 2024-11-29T16:30:04.8490973Z
Size: 4096
Path: C:\Users\pauldeitel\Documents\examples\ch11
Absolute path: C:\Users\pauldeitel\Documents\examples\ch11

Directory contents:
C:\Users\pauldeitel\Documents\examples\ch11\fig11_01
C:\Users\pauldeitel\Documents\examples\ch11\fig11_02
C:\Users\pauldeitel\Documents\examples\ch11\fig11_03
C:\Users\pauldeitel\Documents\examples\ch11\fig11_04
C:\Users\pauldeitel\Documents\examples\ch11\fig11_05
C:\Users\pauldeitel\Documents\examples\ch11\fig11_07
C:\Users\pauldeitel\Documents\examples\ch11\fig11_08
C:\Users\pauldeitel\Documents\examples\ch11\fig11_09
C:\Users\pauldeitel\Documents\examples\ch11\fig11_10
C:\Users\pauldeitel\Documents\examples\ch11\fig11_11
C:\Users\pauldeitel\Documents\examples\ch11\libs
```

```
Enter file or directory name:
c:\Users\pauldeitel\Documents\examples\ch11\fig11_01\FileAndDirectoryInfo.java

FileAndDirectoryInfo.java exists
Is not a directory
Is an absolute path
Last modified: 2024-11-28T20:22:22.8167285Z
Size: 2992
Path: c:\Users\pauldeitel\Documents\examples\ch11\fig11_01\FileAndDirectoryIn-
fo.java
Absolute path: c:\Users\pauldeitel\Documents\examples\ch11\fig11_01\FileAndDi-
rectoryInfo.java
```

```
Path documentsPath =
   Path.of(System.getProperty("user.home"), "Documents");
```

```
System.getProperty("user.home")
```

```
c:\Users\YourAccount\Documents
```

```
1  // Fig. 11.2: CreateTextFile.java
2  // Writing data to a sequential text file with class Formatter.
3  import java.util.Formatter;
4  import java.nio.file.Path;
5  import java.util.List;
6
7  // record class to represent an account
8  record Account(int accountNumber, String name, double balance) {}
9
```

```
10 public class CreateTextFile {
11    public static void main(String[] args) {
12       // account data to write to a text file
13       List<Account> accounts = List.of(
14          new Account(100, "Devi", 24.98),
15          new Account(200, "Taylor", 345.67),
16          new Account(300, "Huber", 0.00),
17          new Account(400, "Ito", -42.16),
18          new Account(500, "Lopez", 224.62)
19       );
20
```

```
        // Path to clients.txt in the user's Documents folder
        Path filePath = Path.of(System.getProperty("user.home"),
           "Documents", "clients.txt");

```

```
        // open clients.txt, output data to the file then close clients.txt
        try (var output = new Formatter(filePath.toString())) {
           // output each Account to file
           for (var account : accounts) {
              output.format("%d %s %.2f%n", account.accountNumber(),
                 account.name(), account.balance());
           }

           System.out.printf(
              "Text file created at: %s%n", filePath.toAbsolutePath());
        }
        catch (Exception e) {
           System.err.printf(
              "Error writing text file: %s%n", e.getMessage());
        }
     }
  }
```

```
Text file created at: C:\Users\pauldeitel\Documents\clients.txt
```

```
  // Fig. 11.3: ReadTextFile.java
  // Reading a text file using a Scanner.
  import java.nio.file.Path;
  import java.util.Scanner;

  public class ReadTextFile {
     public static void main(String[] args) {
        // Path to clients.txt in user's Documents folder
        Path filePath = Path.of(System.getProperty("user.home"),
           "Documents", "clients.txt");

        // open clients.txt, read its contents and close the file
        try (var input = new Scanner(filePath)) {
           System.out.printf("%-10s%-12s%10s%n",
              "Account", "Name", "Balance");

           // read record from file
           while (input.hasNext()) { // while there is more to read
              // display record contents
              System.out.printf("%-10d%-12s%10.2f%n",
                 input.nextInt(), input.next(), input.nextDouble());
           }
        }
        catch (Exception e) {
           System.err.printf(
              "Error reading file: %s%n", e.getMessage());
        }
     }
  }
```

```
Account   Name      Balance
100       Devi        24.98
200       Taylor     345.67
300       Huber        0.00
400       Ito        -42.16
500       Lopez      224.62
```

```
{"accountNumber": 100, "name": "Devi", "balance": 24.98}
```

https://javadoc.io/doc/com.fasterxml.jackson.core/jackson-core/latest

https://github.com/FasterXML/jackson

```java
// Fig. 11.4: CreateJSONFile.java
// Writing data to a JSON file using the Jackson open-source library.
import com.fasterxml.jackson.databind.ObjectMapper;
import java.io.IOException;
import java.nio.file.Path;
import java.util.List;

// record class to represent an account
record Account(int accountNumber, String name, double balance) {}

public class CreateJSONFile {
   public static void main(String[] args) {
      // account data to write to JSON
      List<Account> accounts = List.of(
         new Account(100, "Devi", 24.98),
         new Account(200, "Taylor", 345.67),
         new Account(300, "Huber", 0.00),
         new Account(400, "Ito", -42.16),
         new Account(500, "Lopez", 224.62)
      );

      // Path to clients.json in user's Documents folder
      Path filePath = Path.of(System.getProperty("user.home"),
         "Documents", "clients.json");

      var mapper = new ObjectMapper(); // performs the serialization

      try {
         // Write accounts to a JSON file
         mapper.writeValue(filePath.toFile(), accounts);
         System.out.printf(
            "JSON file created at: %s%n", filePath.toAbsolutePath());
      }
      catch (IOException e) {
         System.err.printf(
            "Error writing JSON file: %s%n", e.getMessage());
      }
   }
}
```

```
JSON file created at: C:\Users\pauldeitel\Documents\clients.json
```

```json
[
  {
    "accountNumber": 100,
    "name": "Devi",
    "balance": 24.98
  },
  {
    "accountNumber": 200,
    "name": "Taylor",
    "balance": 345.67
  },
  {
    "accountNumber": 300,
    "name": "Huber",
    "balance": 0.0
  },
  {
    "accountNumber": 400,
    "name": "Ito",
    "balance": -42.16
  },
  {
    "accountNumber": 500,
    "name": "Lopez",
    "balance": 224.62
  }
]
```

```
javac -cp ".;../libs/*" CreateJSONFile.java
```

```
java -cp ".;../libs/*" CreateJSONFile
```

```java
// Fig. 11.5: ReadJSONFile.java
// Reading data from a JSON file using the Jackson open-source library.
import com.fasterxml.jackson.core.type.TypeReference;
import com.fasterxml.jackson.databind.ObjectMapper;
import java.io.IOException;
import java.nio.file.Path;
import java.util.List;

// record class to represent an account
record Account(int accountNumber, String name, double balance) {}

public class ReadJSONFile {
   public static void main(String[] args) {
      // Path to clients.json in user's Documents folder
      Path filePath = Path.of(System.getProperty("user.home"),
         "Documents", "clients.json");

      var mapper = new ObjectMapper(); // performs the deserialization

      try {
         // Read accounts from the JSON file
         List<Account> accounts = mapper.readValue(
            filePath.toFile(), new TypeReference<List<Account>>() {});

         // display the accounts
         System.out.printf(
            "%-10s %-10s %7s%n", "Account", "Name", "Balance");

         for (var account : accounts) {
            System.out.printf("%-10d %-10s %7.2f%n",
               account.accountNumber(), account.name(),
               account.balance());
         }
      }
      catch (IOException e) {
         System.err.printf(
            "Error reading JSON file: %s%n", e.getMessage());
      }
   }
}
```

```
Account    Name       Balance
100        Devi         24.98
200        Taylor      345.67
300        Huber         0.00
400        Ito         -42.16
500        Lopez       224.62
```

```java
new TypeReference<List<Account>>() {}
```

```java
class MyTypeReference extends TypeReference<List<Account>> {
   // empty class body
}
```

```
https://openweathermap.org/api
```

```
export OPENWEATHERMAP_API_KEY="YourAPIKey"
```

```
{
  "coord": {
    "lon": -71.0598,
    "lat": 42.3584
  },
  "weather": [
    {
      "id": 802,
      "main": "Clouds",
      "description": "scattered clouds",
      "icon": "03d"
    }
  ],
  "base": "stations",
  "main": {
    "temp": 6.55,
    "feels_like": 2.61,
    "temp_min": 5.53,
    "temp_max": 8.06,
    "pressure": 1009,
    "humidity": 57,
    "sea_level": 1009,
    "grnd_level": 1005
  },
  "visibility": 10000,
  "wind": {
    "speed": 6.69,
    "deg": 240,
    "gust": 12.35
  },
  "clouds": {
    "all": 40
  },
  "dt": 1732901160,
  "sys": {
    "type": 2,
    "id": 2013408,
    "country": "US",
    "sunrise": 1732881140,
    "sunset": 1732914812
  },
  "timezone": -18000,
  "id": 4930956,
  "name": "Boston",
  "cod": 200
}
```

```
java -cp ".;../libs/*" WeatherApp Boston
```

```
// Fig. 11.7: WeatherApp.java
// Use an OpenWeatherMap web service to get a city's current weather.
import com.fasterxml.jackson.annotation.JsonIgnoreProperties;
import com.fasterxml.jackson.databind.ObjectMapper;

import java.net.URI;
import java.net.URLEncoder;
import java.net.http.HttpClient;
import java.net.http.HttpRequest;
import java.net.http.HttpResponse;
import java.nio.charset.StandardCharsets;
import java.time.Instant;
import java.time.ZoneId;
import java.time.ZoneOffset;
import java.time.format.DateTimeFormatter;

```

```
17  // record classes for JSON mapping
18  @JsonIgnoreProperties(ignoreUnknown = true)
19  record WeatherData(
20     Weather[] weather,
21     long dt, // date and time
22     Main main,
23     int timezone,
24     String name
25  ) {}
26
```

```
27  @JsonIgnoreProperties(ignoreUnknown = true)
28  record Weather(
29     String main,
30     String description,
31     String icon
32  ) {}
33
34  @JsonIgnoreProperties(ignoreUnknown = true)
35  record Main(
36     double temp,
37     double feels_like,
38     int pressure,
39     int humidity
40  ) {}
41
```

```
42  public class WeatherApp {
43     public static void main(String[] args) throws Exception {
44        // Check for proper command-line arguments
45        if (args.length != 1) {
46           System.out.println("Usage: WeatherApp <city>");
47           System.exit(1);
48        }
49
50        String city = args[0]; // city from command-line argument
51
```

```
52        // load API key from OPENWEATHERMAP_API_KEY environment variable
53        String apiKey = System.getenv("OPENWEATHERMAP_API_KEY");
54
55        // check if API key exists and is set
56        if (apiKey == null || apiKey.isBlank()) {
57           System.out.println(
58              "OPENWEATHERMAP_API_KEY environment variable is not set.");
59           System.exit(1);
60        }
61
```

```
62        // URL encode the city parameter
63        String encodedCity =
64           URLEncoder.encode(city, StandardCharsets.UTF_8);
65
66        // create the API URL as a String
67        String apiUrl = String.format(
68            "https://api.openweathermap.org/" +
69            "data/2.5/weather?q=%s&units=metric&appid=%s",
70            encodedCity, apiKey);
71
```

```
72          // use Java's HttpClient to make the request
73          HttpClient client = HttpClient.newHttpClient();
74
75          // build the request that HttpClient will use to
76          // invoke the OpenWeatherMap web service
77          HttpRequest request =
78             HttpRequest.newBuilder()
79                .uri(URI.create(apiUrl))
80                .GET()
81                .build();
82
83          // send the HTTP request and get the response
84          HttpResponse<String> response =
85             client.send(request, HttpResponse.BodyHandlers.ofString());
86
87          // check if the request was successful
88          if (response.statusCode() != 200) {
89             System.out.printf("Error: Received HTTP status %d\n",
90                response.statusCode());
91             System.exit(1);
92          }
93
```

```
94          // parse the JSON response directly into the WeatherData record
95          ObjectMapper mapper = new ObjectMapper();
96          WeatherData weatherData =
97             mapper.readValue(response.body(), WeatherData.class);
98
```

```
99          // convert temperatures to Fahrenheit
100         double tempCurrentF = (weatherData.main().temp() * 9 / 5) + 32;
101         double feelsLikeF =
102            (weatherData.main().feels_like() * 9 / 5) + 32;
103
```

```
104         // generate the icon URL
105         String iconName = weatherData.weather()[0].icon();
106         String iconUrl = String.format(
107            "https://openweathermap.org/img/wn/%s@4x.png", iconName);
108
```

```
109         // format the date and time based on the response
110         Instant timestamp = Instant.ofEpochSecond(weatherData.dt());
111         String formattedDateTime =
112            DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss")
113               .withZone(ZoneId.ofOffset("UTC",
114                  ZoneOffset.ofTotalSeconds(weatherData.timezone())))
115               .format(timestamp);
116
```

```
        System.out.printf("%s Weather\n", weatherData.name());
        System.out.printf("Date and Time: %s\n", formattedDateTime);
        System.out.printf("Temperature: %.1f C (%.1f F)\n",
           weatherData.main().temp(), tempCurrentF);
        System.out.printf("Feels like: %.1f C (%.1f F)\n",
           weatherData.main().feels_like(), feelsLikeF);
        System.out.printf("Pressure: %d hPa\n",
           weatherData.main().pressure());
        System.out.printf("Humidity: %d%%\n",
           weatherData.main().humidity());
        System.out.printf("Conditions: %s\n",
           weatherData.weather()[0].description());
        System.out.printf("Icon: %s\n", iconUrl);
    }
}
```

```
Boston Weather
Date and Time: 2024-12-28 10:18:25
Temperature: 3.8 C (38.9 F)
Feels like: 1.5 C (34.6 F)
Pressure: 1026 hPa
Humidity: 89%
Conditions: light rain
Icon: https://openweathermap.org/img/wn/10d@4x.png
```

https://github.com/awesomedata/awesome-public-datasets
https://vincentarelbundock.github.io/Rdatasets/articles/data.html

https://catalog.data.gov/dataset/?res_format=CSV

```
// Fig. 11.8: CreateCSVFile.java
// Writing data to a CSV file using the Jackson open-source library.
import com.fasterxml.jackson.databind.ObjectWriter;
import com.fasterxml.jackson.dataformat.csv.CsvMapper;
import com.fasterxml.jackson.dataformat.csv.CsvSchema;
import java.io.IOException;
import java.nio.file.Path;
import java.util.List;

// record class to represent an account
record Account(int accountNumber, String name, double balance) {}

public class CreateCSVFile {
   public static void main(String[] args) {
      // account data to write to
      List<Account> accounts = List.of(
         new Account(100, "Devi", 24.98),
         new Account(200, "Taylor", 345.67),
         new Account(300, "Huber", 0.00),
         new Account(400, "Ito", -42.16),
         new Account(500, "Lopez", 224.62)
      );
```

```
23
24          // Path to clients.csv in user's Documents folder
25          Path filePath = Path.of(System.getProperty("user.home"),
26             "Documents", "clients.csv");
27
28          // create Jackson CsvMapper and Schema
29          var mapper = new CsvMapper();
30          CsvSchema schema =
31             mapper.schemaFor(Account.class) // determines CSV files's fields
32                .withHeader(); // first line in clients.csv is the field names
33
34          try {
35             // ObjectWriter writes data to CSV file using specified schema
36             ObjectWriter writer = mapper.writer(schema);
37             writer.writeValue(filePath.toFile(), accounts);
38             System.out.printf(
39                "CSV file created at: %s%n", filePath.toAbsolutePath());
40          }
41          catch (IOException e) {
42             System.err.printf(
43                "Error writing CSV file: %s%n", e.getMessage());
44          }
45       }
46    }
```

```
CSV file created at: C:\Users\pauldeitel\Documents\clients.csv
```

```
1  // Fig. 11.9: ReadCSVFile.java
2  // Reading data from a CSV file using the Jackson open-source library.
3  import com.fasterxml.jackson.databind.MappingIterator;
4  import com.fasterxml.jackson.dataformat.csv.CsvMapper;
5  import com.fasterxml.jackson.dataformat.csv.CsvSchema;
6
7  import java.io.IOException;
8  import java.nio.file.Path;
9  import java.util.List;
10
11 // record class to represent an account
12 record Account(int accountNumber, String name, double balance) {}
13
14 public class ReadCSVFile {
15    public static void main(String[] args) {
16       // Path to clients.csv in user's Documents folder
17       Path filePath = Path.of(System.getProperty("user.home"),
18          "Documents", "clients.csv");
19
20       var mapper = new CsvMapper(); // reads CSV records
21
22       // specify that Jackson should map each CSV column to
23       // an instance variable in the record class Account
24       CsvSchema schema =
25          mapper.schemaFor(Account.class) // fields map to Account objects
26                .withHeader(); // first line of text is the field names
27
28       // read the CSV file
29       try {
30          // MappingIterator provides access to records as Account objects
```

```
            MappingIterator<Account> iterator =
               mapper.readerFor(Account.class).with(schema).readValues(
                  filePath.toFile());

            List<Account> accounts = iterator.readAll();

            // display the accounts
            System.out.printf(
               "%-10s %-10s %7s%n", "Account", "Name", "Balance");

            for (var account : accounts) {
               System.out.printf("%-10d %-10s %7.2f%n",
                  account.accountNumber(), account.name(),
                  account.balance());
            }
         }
         catch (IOException e) {
            System.err.printf(
               "Error reading CSV file: %s%n", e.getMessage());
         }
      }
   }
```

```
Account    Name       Balance
100        Devi         24.98
200        Taylor      345.67
300        Huber         0.00
400        Ito         -42.16
500        Lopez       224.62
```

https://vincentarelbundock.github.io/Rdatasets/csv/carData/TitanicSurvival.csv

https://hbiostat.org/data/repo/titanic

```
// Fig. 11.10: TitanicAnalysis.java
// Reading the Titanic dataset from a CSV file, then analyzing it.
import com.fasterxml.jackson.annotation.JsonIgnoreProperties;
import com.fasterxml.jackson.databind.MappingIterator;
import com.fasterxml.jackson.dataformat.csv.CsvMapper;
import com.fasterxml.jackson.dataformat.csv.CsvSchema;
import java.nio.file.Path;
import java.util.ArrayList;
import java.util.List;

```

```
public class TitanicAnalysis {
   // record class representing the fields in each dataset row
   @JsonIgnoreProperties(ignoreUnknown = true)
   public record TitanicRecord(
      String survived,
      String sex,
      String age,
      String passengerClass
   ) {}

```

```
    public static void main(String[] args) throws Exception {
       var mapper = new CsvMapper(); // reads CSV records

       // specify that Jackson should map each CSV column
       // to an object based on the column names
       CsvSchema schema = CsvSchema.emptySchema().withHeader();

       // create the MappingIterator that knows how to read
       // the file's records into TitanicRecord objects
       MappingIterator<TitanicRecord> iterator =
          mapper.readerFor(TitanicRecord.class).with(schema).readValues(
             Path.of("TitanicSurvival.csv").toFile());

       // load dataset into a List of TitanicRecord objects
       List<TitanicRecord> titanic = iterator.readAll();

```

```
       // create lists representing each column with the correct type
       var survived = new ArrayList<String>();
       var sex = new ArrayList<String>();
       var age = new ArrayList<Double>();
       var pclass = new ArrayList<Integer>();

       // populate the column lists
       for (TitanicRecord record : titanic) {
          survived.add(record.survived());
          sex.add(record.sex());
          age.add(!record.age().isEmpty() ?
             Double.parseDouble(record.age()) : Double.NaN);
          pclass.add(
             switch (record.passengerClass()) {
                case "1st" -> 1;
                case "2nd" -> 2;
                case "3rd" -> 3;
                default -> throw new IllegalArgumentException(
                   "Unknown class: " + record.passengerClass());
             }
          );
       }

```

```
       // display first 5 rows
       System.out.printf("First five rows:%n%-10s%-8s%-6s%s%n",
          "survived", "sex", "age", "class");

       for (int i = 0; i < 5; ++i) {
          System.out.printf("%-10s%-8s%-6.1f%d%n",
             survived.get(i), sex.get(i), age.get(i), pclass.get(i));
       }

```

```
First five rows:
survived  sex     age   class
yes       female  29.0  1
yes       male    0.9   1
no        female  2.0   1
no        male    30.0  1
no        female  25.0  1
```

```
69          // display last 5 rows
70          System.out.printf("%nLast five rows:%n%-10s%-8s%-6s%s%n",
71             "survived", "sex", "age", "class");
72          int count = titanic.size();
73
74          for (int i = count - 5; i < count; ++i) {
75             System.out.printf("%-10s%-8s%-6.1f%d%n",
76                survived.get(i), sex.get(i), age.get(i), pclass.get(i));
77          }
78
```

```
Last five rows:
survived  sex     age   class
no        female  14.5  3
no        female  NaN   3
no        male    26.5  3
no        male    27.0  3
no        male    29.0  3
```

```
79          // clean the age column (remove NaN)
80          var cleanAge = new ArrayList<Double>();
81          for (Double value : age) {
82             if (!value.isNaN()) {
83                cleanAge.add(value);
84             }
85          }
86
```

```
87          // descriptive statistics for cleaned age column
88          cleanAge.sort(null); // helps determine min, max and median ages
89          int size = cleanAge.size();
90
91          // get median age
92          double medianAge = (size % 2 == 0) ?
93             (cleanAge.get(size / 2 - 1) + cleanAge.get(size / 2)) / 2.0 :
94             cleanAge.get(size / 2);
95
96          // calculate average of ages
97          double sum = 0.0;
98
99          for (double value : cleanAge) {
100            sum += value;
101         }
102
103         double averageAge = sum / size;
104
```

```
105         // display the age statistics
106         System.out.printf("%nDescriptive statistics for the age column:%n");
107         System.out.printf("Passengers with age data: %d%n", size);
108         System.out.printf("Average age: %.2f%n", averageAge);
109         System.out.printf("Minimum age: %.2f%n", cleanAge.getFirst());
110         System.out.printf("Maximum age: %.2f%n", cleanAge.getLast());
111         System.out.printf("Median age: %.2f%n", medianAge);
112
```

```
Descriptive statistics for the age column:
Passengers with age data: 1046
Average age: 29.88
Minimum age: 0.17
Maximum age: 80.00
Median age: 28.00
```

```
113         // passenger counts by class
114         int[] counts = new int[4]; // ignore element 0
115
116         for (int value : pclass) {
117            ++counts[value];
118         }
119
120         System.out.printf(
121            "%nPassenger counts by class:%n1st: %d%n2nd: %d%n3rd: %d%n",
122            counts[1], counts[2], counts[3]);
123
```

```
Passenger counts by class:
1st: 323
2nd: 277
3rd: 709
```

```
124         // summarize counts of people who survived by sex and by class
125         int survivorCount = 0;
126         int[] survivorsBySex = new int[2];
127         int[] survivorsByClass = new int[4]; // ignore element 0
128
129         for (int i = 0; i < survived.size(); ++i) {
130            if (survived.get(i).equals("yes")) {
131               ++survivorCount;
132
133               if (sex.get(i).equals("female")) {
134                  ++survivorsBySex[0];
135               }
136               else {
137                  ++survivorsBySex[1];
138               }
139
140               ++survivorsByClass[pclass.get(i)];
141            }
142         }
143
```

```
144         // calculate and display the survivor statistics
145         System.out.printf("%nSurvived count: %d%nDied count: %d%n",
146            survivorCount, survived.size() - survivorCount);
147         System.out.printf("Percent who survived: %.2f%%%n",
148            100.0 * survivorCount / survived.size());
149         System.out.printf("Female survivor percentage: %.2f%%%n",
150            100.0 * survivorsBySex[0] / survivorCount);
151         System.out.printf("Male survivor percentage: %.2f%%%n",
152            100.0 * survivorsBySex[1] / survivorCount);
153         System.out.printf("1st class survivor percentage: %.2f%%%n",
154            100.0 * survivorsByClass[1] / survivorCount);
155         System.out.printf("2nd class survivor percentage: %.2f%%%n",
156            100.0 * survivorsByClass[2] / survivorCount);
157         System.out.printf("3rd class survivor percentage: %.2f%%%n",
158            100.0 * survivorsByClass[3] / survivorCount);
159      }
160   }
```

```
Survived count: 500
Died count: 809
Percent who survived: 38.20%
Female survivor percentage: 67.80%
Male survivor percentage: 32.20%
1st class survivor percentage: 40.00%
2nd class survivor percentage: 23.80%
3rd class survivor percentage: 36.20%
```

```java
// Fig. 11.11: CryptographyDemo.java
// Encrypting and decrypting messages with AES,
// and using RSA to encrypt and decrypt the AES key.
import javax.crypto.Cipher;
import javax.crypto.KeyGenerator;
import javax.crypto.SecretKey;
import javax.crypto.spec.SecretKeySpec;
import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.security.PublicKey;
import java.security.PrivateKey;
import java.security.SecureRandom;
import java.util.Base64;
import java.util.Scanner;

public class CryptographyDemo {
   public static void main(String[] args) {
      var scanner = new Scanner(System.in);

      System.out.println("Enter the text to encrypt:");
      String plaintext = scanner.nextLine();

      System.out.println(
         "\nEnter the secret key seed (for reproducibility):");
      String seedString = scanner.nextLine();

      try { // encryption/decryption might throw exceptions
         // generate a reproducible AES key from the seedString
         SecretKey secretKey = generateKey(seedString);
         System.out.printf("%nSecretKey:%n%s%n",
            Base64.getEncoder().encodeToString(secretKey.getEncoded()));

         // AES encrypt the plaintext
         String ciphertext = encrypt(plaintext, secretKey);
         System.out.printf("%nEncrypted message:%n%s%n", ciphertext);
```

```
Enter the text to encrypt:
Welcome to encryption and decryption using AES and RSA in Java!

Enter the secret key seed (for reproducibility):
XMWUJBVYHXZ

SecretKey:
8ew8jBJVSpkRtG4zorwxd34JBpUxEdOt+ec0QTfr7cE=

Encrypted message:
Cz9o3YoFcX+yebfAOuMPMi+UZz2bc5jJdstfzapaNKyvWTHQf2nmPHQ9SjRVue1Qc179qM-
wGI44tpOHKjErKrA==
```

```java
         // generate RSA key pair -- normally done by message receiver
         var keyPairGenerator = KeyPairGenerator.getInstance("RSA");
         keyPairGenerator.initialize(2048); // set key size to 2048 bits
         KeyPair rsaKeyPair = keyPairGenerator.generateKeyPair();

         // get RSA public and private keys -- recipient sends the
         // public key to anyone who needs to send them something securely
         PublicKey publicKey = rsaKeyPair.getPublic();
         PrivateKey privateKey = rsaKeyPair.getPrivate();

         // encrypt the AES secret key with the RSA public key;
         // this would be sent to recipient with the encrypted message
         String encryptedAESKey =
            encryptRSA(secretKey.getEncoded(), publicKey);
         System.out.printf(
            "%nRSA Encrypted AES SecretKey:%n%s%n", encryptedAESKey);
```

```
RSA Encrypted AES SecretKey:
RzZ2APUmXjwjWnKiCptMifxdwgADNs9k3qXEbUtTivXfD/kaUd3rExhgfTiBqnbIYdTuuze3C5xxy
l66AEf3Hu7bPOrJbPjbp3yhOD6e/x5Gf//UA+b4d1GM+dOpMAXOF6Oiu658W33M+PTCowDdOyj9xE
8yYsdtW3E4/uJ2CQMt22+87LbbtDwFVqT8x91ZWuqg+gKs7RhiybdZ7z8JzjmEI+Wr5UHOAXT7qvj
SQjZGTKtOuZfJniwPT4isJWHtPPwX1UKwk7GtG5vlnOWgopYuO9bmee2xBVMc/LUzbOpeEIs1J435
bw1bYoaeJiIjVyiyWyjcvsW6btJPNHXZeg==
```

```
            // receiver decrypts AES secret key using RSA private key
            SecretKey decryptedSecretKey =
               decryptRSA(encryptedAESKey, privateKey);
            System.out.printf("%nDecrypted AES SecretKey:%n%s%n",
               Base64.getEncoder().encodeToString(
                  decryptedSecretKey.getEncoded()));

```

```
Decrypted AES SecretKey:
8ew8jBJVSpkRtG4zorwxd34JBpUxEdOt+ec0QTfr7cE=
```

```
            // AES decrypt the text
            System.out.printf("%nDecrypted:%n%s%n",
               decrypt(ciphertext, decryptedSecretKey));
         }
         catch (Exception ex) {
            System.out.printf("""
               An exception occurred during encryption/decryption:
               %s%n
               """, ex);
         }
      }

```

```
Decrypted:
Welcome to encryption and decryption using AES and RSA in Java!
```

```
      // Generate and return a SecretKey. Note: We are seeding a
      // SecureRandom object for reproducibility so you get the same results
      // we show in this program's output. Don't do this in production code.
      public static SecretKey generateKey(String seed) throws Exception {
         // get a secure random number generator
         var random = SecureRandom.getInstance("SHA1PRNG");
         random.setSeed(seed.getBytes()); // for demo reproducibility

         // get an AES SecretKey generator
         var keyGen = KeyGenerator.getInstance("AES");
         keyGen.init(256, random); // set up for 256-bit keys
         return keyGen.generateKey(); // generate and return new SecretKey
      }

      // encrypt a String using AES
      public static String encrypt(
         String plaintext, SecretKey secretKey) throws Exception {

         var cipher = Cipher.getInstance("AES"); // get a Cipher object
         cipher.init(Cipher.ENCRYPT_MODE, secretKey); // configure to encrypt

         // encrypt the byte[] representation of plaintext
         byte[] encrypted = cipher.doFinal(plaintext.getBytes());

         // Base64 encode encrypted as a ciphertext String and return
         return Base64.getEncoder().encodeToString(encrypted);
      }

      // decrypt an AES-encrypted String
      public static String decrypt(
         String ciphertext, SecretKey secretKey) throws Exception {

         var cipher = Cipher.getInstance("AES"); // get a Cipher object
         cipher.init(Cipher.DECRYPT_MODE, secretKey); // configure to decrypt

         // Base64 decode ciphertext to byte array
         byte[] ciphertextBytes = Base64.getDecoder().decode(ciphertext);

         // create plaintext String from ciphertextBytes and return it
         return new String(cipher.doFinal(ciphertextBytes));
      }

```

```
115     // encrypt data using RSA public key -- the result of this method
116     // would be sent to the AES encrypted message's receiver
117     public static String encryptRSA(byte[] data, PublicKey publicKey)
118        throws Exception {
119
120        var cipher = Cipher.getInstance("RSA"); // create RSA Cipher object
121
122        // initialize cipher in encryption mode with recipient's public key
123        cipher.init(Cipher.ENCRYPT_MODE, publicKey);
124
125        // encrypt data (the AES key) using the RSA public key
126        byte[] encryptedData = cipher.doFinal(data);
127
128        // Base64 encode the encrypted AES key
129        return Base64.getEncoder().encodeToString(encryptedData);
130     }
131
```

```
132     // decrypt data using RSA private key and return the SecretKey
133     public static SecretKey decryptRSA(
134        String encryptedData, PrivateKey privateKey) throws Exception {
135
136        var cipher = Cipher.getInstance("RSA"); // create RSA Cipher object
137
138        // initialize cipher in decryption mode with recipient's private key
139        cipher.init(Cipher.DECRYPT_MODE, privateKey);
140
141        // decode the encrypted AES key from Base64 format and decrypt it
142        byte[] decryptedData =
143           cipher.doFinal(Base64.getDecoder().decode(encryptedData));
144
145        // convert decrypted byte array back to an AES SecretKey object
146        return new SecretKeySpec(
147           decryptedData, 0, decryptedData.length, "AES");
148     }
149  }
```

https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/security/interfaces/package-summary.html

https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/util/Collection.html

```
Integer integerObject = 10; // boxes 10 to an Integer object

int value = integerObject; // unboxes 10 from integerObject
```

https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/util/List.html

```
1  // Fig. 12.1: CollectionTest.java
2  // Collection interface demonstrated via an ArrayList object.
3  import java.util.List;
4  import java.util.ArrayList;
5  import java.util.Collection;
6  import java.util.Iterator;
7
8  public class CollectionTest {
9     public static void main(String[] args) {
```

```
10       // add elements in colors array to colorList
11       String[] colors = {"MAGENTA", "RED", "WHITE", "BLUE", "CYAN"};
12       List<String> colorList = new ArrayList<String>();
13
14       for (String color : colors) {
15          colorList.add(color); // adds color to end of colorList
16       }
17
18       // add elements in removeColors array to removeList
19       String[] removeColors = {"RED", "WHITE", "BLUE"};
20       List<String> removeList = new ArrayList<String>();
21
22       for (String color : removeColors) {
23          removeList.add(color);
24       }
25
```

```
26       // output colorList contents
27       System.out.println("ArrayList: ");
28
29       for (int count = 0; count < colorList.size(); ++count) {
30          System.out.printf("%s ", colorList.get(count));
31       }
32
```

```
ArrayList:
MAGENTA RED WHITE BLUE CYAN
```

```
33       // remove from colorList the colors contained in removeList
34       removeColors(colorList, removeList);
35
36       // output colorList contents
37       System.out.printf("%n%nArrayList after calling removeColors:%n");
38
39       for (String color : colorList) {
40          System.out.printf("%s ", color);
41       }
42    }
43
```

```
ArrayList after calling removeColors:
MAGENTA CYAN
```

```
44      // remove colors specified in itemsToRemove from allItems
45      private static void removeColors(Collection<String> allItems,
46         Collection<String> itemsToRemove) {
47
48         Iterator<String> iterator = allItems.iterator(); // get iterator
49
50         // loop while collection has items
51         while (iterator.hasNext()) {
52            if (itemsToRemove.contains(iterator.next())) {
53               iterator.remove(); // remove current element
54            }
55         }
56      }
57   }
```

```
colorList.removeAll(removeList);
```

```
List<String> list = new ArrayList<>();
```

```
1   // Fig. 12.2: ListTest.java
2   // Lists, LinkedLists and ListIterators.
3   import java.util.List;
4   import java.util.LinkedList;
5   import java.util.ListIterator;
6
7   public class ListTest {
8      public static void main(String[] args) {
```

```
9         // add colors elements to list1
10        String[] colors =
11           {"black", "yellow", "green", "blue", "violet", "silver"};
12        List<String> list1 = new LinkedList<>();
13
14        for (String color : colors) {
15           list1.add(color);
16        }
17
18        // add colors2 elements to list2
19        String[] colors2 =
20           {"gold", "white", "brown", "blue", "gray", "silver"};
21        List<String> list2 = new LinkedList<>();
22
23        for (String color : colors2) {
24           list2.add(color);
25        }
26
```

```
27        list1.addAll(list2); // concatenate lists
28
29        printList(list1);
30
```

```
list:
black yellow green blue violet silver gold white brown blue gray silver
```

```
31        convertToUppercaseStrings(list1);
32        printList(list1);
33
```

```
list:
BLACK YELLOW GREEN BLUE VIOLET SILVER GOLD WHITE BROWN BLUE GRAY SILVER
```

```
34          System.out.printf("%nDeleting elements 4 to 6...");
35          removeItems(list1, 4, 7); // remove items 4-6 from list
36          printList(list1);
37          printReversedList(list1);
38       }
39
```

```
Deleting elements 4 to 6...
list:
BLACK YELLOW GREEN BLUE WHITE BROWN BLUE GRAY SILVER

Reversed list:
SILVER GRAY BLUE BROWN WHITE BLUE GREEN YELLOW BLACK
```

```
40       // output List contents
41       private static void printList(List<String> list) {
42          System.out.printf("%nlist:%n");
43
44          for (String color : list) {
45             System.out.printf("%s ", color);
46          }
47
48          System.out.println();
49       }
50
```

```
51       // locate String objects and convert to uppercase
52       private static void convertToUppercaseStrings(List<String> list) {
53          ListIterator<String> iterator = list.listIterator();
54
55          while (iterator.hasNext()) {
56             String color = iterator.next(); // get item
57             iterator.set(color.toUpperCase()); // convert to upper case
58          }
59       }
60
```

```
61       // obtain sublist and use clear method to delete sublist items
62       private static void removeItems(List<String> list,
63          int start, int end) {
64          list.subList(start, end).clear(); // remove items
65       }
66
```

```
67       // print reversed list
68       private static void printReversedList(List<String> list) {
69          ListIterator<String> iterator = list.listIterator(list.size());
70
71          System.out.printf("%nReversed list:%n");
72
73          // print list in reverse order
74          while (iterator.hasPrevious()) {
75             System.out.printf("%s ", iterator.previous());
76          }
77       }
78    }
```

```java
// Fig. 12.3: UsingToArray.java
// Viewing arrays as Lists and converting Lists to arrays.
import java.util.LinkedList;
import java.util.Arrays;

public class UsingToArray {
   public static void main(String[] args) {
      String[] colors = {"black", "blue", "yellow"};
      LinkedList<String> list = new LinkedList<>(Arrays.asList(colors));

      list.addLast("red"); // add as last item
      list.add("pink"); // add to the end
      list.add(3, "green"); // add at 3rd index
      list.addFirst("cyan"); // add as first item

      // get LinkedList elements as an array
      colors = list.toArray(new String[list.size()]);

      System.out.println("colors: ");

      for (String color : colors) {
         System.out.println(color);
      }
   }
}
```

```
colors:
cyan
black
blue
yellow
green
red
pink
```

```java
// Fig. 12.4: Sort1.java
// Collections method sort.
import java.util.List;
import java.util.Arrays;
import java.util.Collections;

public class Sort1 {
   public static void main(String[] args) {
      String[] suits = {"Hearts", "Diamonds", "Clubs", "Spades"};

      // Create and display a list containing the suits array elements
      List<String> list = Arrays.asList(suits);
      System.out.printf("Unsorted array elements: %s%n", list);

      Collections.sort(list); // sort ArrayList
      System.out.printf("Sorted array elements: %s%n", list);
   }
}
```

```
Unsorted array elements: [Hearts, Diamonds, Clubs, Spades]
Sorted array elements: [Clubs, Diamonds, Hearts, Spades]
```

```java
// Fig. 12.5: Sort2.java
// Using a Comparator object with method sort.
import java.util.List;
import java.util.Arrays;
import java.util.Collections;

public class Sort2 {
   public static void main(String[] args) {
      String[] suits = {"Hearts", "Diamonds", "Clubs", "Spades"};

      // Create and display a list containing the suits array elements
      List<String> list = Arrays.asList(suits); // create List
      System.out.printf("Unsorted array elements: %s%n", list);

      // sort in descending order using a comparator
      Collections.sort(list, Collections.reverseOrder());
      System.out.printf("Sorted list elements: %s%n", list);
   }
}
```

```
Unsorted array elements: [Hearts, Diamonds, Clubs, Spades]
Sorted list elements: [Spades, Hearts, Diamonds, Clubs]
```

```java
// Fig. 12.6: TimeComparator.java
// Custom Comparator class that compares two Time2 objects.
import java.util.Comparator;

public class TimeComparator implements Comparator<Time2> {
   @Override
   public int compare(Time2 time1, Time2 time2) {
      int hourDifference = time1.getHour() - time2.getHour();

      if (hourDifference != 0) { // test the hour first
         return hourDifference;
      }

      int minuteDifference = time1.getMinute() - time2.getMinute();

      if (minuteDifference != 0) { // then test the minute
         return minuteDifference;
      }

      int secondDifference = time1.getSecond() - time2.getSecond();
      return secondDifference;
   }
}
```

```java
// Fig. 12.7: Sort3.java
// Collections method sort with a custom Comparator object.
import java.util.List;
import java.util.ArrayList;
import java.util.Collections;

public class Sort3 {
   public static void main(String[] args) {
      List<Time2> list = new ArrayList<>(); // create List

      list.add(new Time2(6, 24, 34));
      list.add(new Time2(18, 14, 58));
      list.add(new Time2(6, 5, 34));
      list.add(new Time2(12, 14, 58));
      list.add(new Time2(6, 24, 22));

      // output List elements
      System.out.printf("Unsorted array elements:%n%s%n", list);

      // sort in order using a comparator
      Collections.sort(list, new TimeComparator());

      // output List elements
      System.out.printf("Sorted list elements:%n%s%n", list);
   }
}
```

```
Unsorted array elements:
[6:24:34 AM, 6:14:58 PM, 6:05:34 AM, 12:14:58 PM, 6:24:22 AM]
Sorted list elements:
[6:05:34 AM, 6:24:22 AM, 6:24:34 AM, 12:14:58 PM, 6:14:58 PM]
```

```java
// Fig. 12.8: DeckOfCards.java
// Card shuffling and dealing with Collections method shuffle.
import java.util.ArraysList;
import java.util.List;
import java.util.Collections;

// class to represent a Card in a deck of cards
class Card {
   public enum Face {Ace, Deuce, Three, Four, Five, Six,
      Seven, Eight, Nine, Ten, Jack, Queen, King }
   public enum Suit {Clubs, Diamonds, Hearts, Spades}

   private final Face face;
   private final Suit suit;

   // constructor
   public Card(Face face, Suit suit) {
       this.face = face;
       this.suit = suit;
   }

   // return face of the card
   public Face getFace() {return face;}

   // return suit of Card
   public Suit getSuit() {return suit;}

   // return String representation of Card
   public String toString() {
      return String.format("%s of %s", face, suit);
   }
}

// class DeckOfCards declaration
public class DeckOfCards {
   private List<Card> deck = new ArrayList<>(); // List to store Cards

   // set up deck of Cards and shuffle
```

```
39      public DeckOfCards() {
40         // populate deck with Card objects
41         for (Card.Suit suit : Card.Suit.values()) {
42            for (Card.Face face : Card.Face.values()) {
43               deck.add(new Card(face, suit));
44            }
45         }
46
47         Collections.shuffle(deck);  // shuffle deck
48      }
49
50      // output deck
51      public void printCards() {
52         // display 52 cards in four columns
53         for (int i = 0; i < deck.size(); ++i) {
54            System.out.printf("%-19s%s", deck.get(i),
55               ((i + 1) % 4 == 0) ? System.lineSeparator() : "");
56         }
57      }
58
59      public static void main(String[] args) {
60         DeckOfCards cards = new DeckOfCards();
61         cards.printCards();
62      }
63   }
```

```
Deuce of Clubs     Six of Spades      Nine of Diamonds   Ten of Hearts
Three of Diamonds  Five of Clubs      Deuce of Diamonds  Seven of Clubs
Three of Spades    Six of Diamonds    King of Clubs      Jack of Hearts
Ten of Spades      King of Diamonds   Eight of Spades    Six of Hearts
Nine of Clubs      Ten of Diamonds    Eight of Diamonds  Eight of Hearts
Ten of Clubs       Five of Hearts     Ace of Clubs       Deuce of Hearts
Queen of Diamonds  Ace of Diamonds    Four of Clubs      Nine of Hearts
Ace of Spades      Deuce of Spades    Ace of Hearts      Jack of Diamonds
Seven of Diamonds  Three of Hearts    Four of Spades     Four of Diamonds
Seven of Spades    King of Hearts     Seven of Hearts    Five of Diamonds
Eight of Clubs     Three of Clubs     Queen of Clubs     Queen of Spades
Six of Clubs       Nine of Spades     Four of Hearts     Jack of Clubs
Five of Spades     King of Spades     Jack of Spades     Queen of Hearts
```

```
1  // Fig. 12.9: Algorithms1.java
2  // Collections methods reverse, fill, copy, max and min.
3  import java.util.List;
4  import java.util.Arrays;
5  import java.util.Collections;
6
7  public class Algorithms1 {
8     public static void main(String[] args) {
9        // create and display a List<Character>
10       Character[] letters = {'P', 'C', 'M'};
11       List<Character> list = Arrays.asList(letters); // get List
12       outputList(list);
13
14       // reverse and display the List<Character>
15       Collections.reverse(list); // reverse order the elements
16       System.out.printf("%nAfter calling reverse, list contains:%n");
17       outputList(list);
18
19       // create copyList from an array of 3 Characters
20       Character[] lettersCopy = new Character[3];
21       List<Character> copyList = Arrays.asList(lettersCopy);
22
23       // copy the contents of list into copyList
24       Collections.copy(copyList, list);
25       System.out.printf("%nAfter copying, copyList contains:%n");
26       outputList(copyList);
```

```
27
28          // fill list with Rs
29          Collections.fill(list, 'R');
30          System.out.printf("%nAfter calling fill, list contains:%n");
31          outputList(list);
32       }
33
34       // output List information
35       private static void outputList(List<Character> list) {
36          System.out.printf("The list is: %s%n", list);
37          System.out.printf("Max: %s; Min: %s%n", Collections.max(list),
38             Collections.min(list));
39       }
40    }
```

```
The list is: [P, C, M]
Max: P; Min: C

After calling reverse, list contains:
The list is: [M, C, P]
Max: P; Min: C

After copying, copyList contains:
The list is: [M, C, P]
Max: P; Min: C

After calling fill, list contains:
The list is: [R, R, R]
Max: R; Min: R
```

```
1  // Fig. 12.10: BinarySearchTest.java
2  // Collections method binarySearch.
3  import java.util.List;
4  import java.util.Arrays;
5  import java.util.Collections;
6  import java.util.ArrayList;
7
8  public class BinarySearchTest {
9     public static void main(String[] args) {
10       // create an ArrayList<String> from the contents of colors array
11       String[] colors = {"red", "white", "blue", "black", "yellow",
12          "purple", "tan", "pink"};
13       List<String> list = new ArrayList<>(Arrays.asList(colors));
14
15       Collections.sort(list); // sort the ArrayList
16       System.out.printf("Sorted ArrayList: %s%n", list);
17
18       // search list for various values
19       printSearchResults(list, "black");
20       printSearchResults(list, "red");
21       printSearchResults(list, "pink");
22       printSearchResults(list, "aqua"); // below lowest
23       printSearchResults(list, "gray"); // does not exist
24       printSearchResults(list, "teal"); // does not exist
25    }
26
```

```
   // perform search and display result
   private static void printSearchResults(
      List<String> list, String key) {

      System.out.printf("%nSearching for: %s%n", key);
      int result = Collections.binarySearch(list, key);

      if (result >= 0) {
         System.out.printf("Found at index %d%n", result);
      }
      else {
         System.out.printf("Not Found (%d)%n",result);
      }
   }
}
```

```
Sorted ArrayList: [black, blue, pink, purple, red, tan, white, yellow]

Searching for: black
Found at index 0

Searching for: red
Found at index 4

Searching for: pink
Found at index 2

Searching for: aqua
Not Found (-1)

Searching for: gray
Not Found (-3)

Searching for: teal
Not Found (-7)
```

```
// Fig. 12.11: Algorithms2.java
// Collections methods addAll, frequency and disjoint.
import java.util.ArrayList;
import java.util.List;
import java.util.Arrays;
import java.util.Collections;

public class Algorithms2 {
   public static void main(String[] args) {
      // initialize list1 and list2
      String[] colors = {"red", "white", "yellow", "blue"};
      List<String> list1 = Arrays.asList(colors);
      List<String> list2 = new ArrayList<>();

      list2.add("black"); // add "black" to the end of list2
      list2.add("red"); // add "red" to the end of list2
      list2.add("green"); // add "green" to the end of list2

      System.out.printf("Before addAll, list2 contains: %s%n", list2);
      Collections.addAll(list2, colors); // add colors Strings to list2
      System.out.printf("After addAll, list2 contains: %s%n", list2);

      // get frequency of "red"
      int frequency = Collections.frequency(list2, "red");
      System.out.printf("%nFrequency of red in list2: %d%n", frequency);

      // check whether list1 and list2 have any elements in common
      boolean areDisjoint = Collections.disjoint(list1, list2);

      System.out.printf("list1 and list2 %s elements in common%n",
         (areDisjoint ? "do not have" : "have"));
   }
}
```

```
Before addAll, list2 contains: [black, red, green]
After addAll, list2 contains: [black, red, green, red, white, yellow, blue]

Frequency of red in list2: 2
list1 and list2 have elements in common
```

```java
// Fig. 12.12: PriorityQueueTest.java
// PriorityQueue test program.
import java.util.PriorityQueue;

public class PriorityQueueTest {
   public static void main(String[] args) {
      // queue of capacity 11
      var queue = new PriorityQueue<Double>();

      // insert elements to queue
      queue.offer(3.2);
      queue.offer(9.8);
      queue.offer(5.4);

      System.out.print("Polling from queue: ");

      // display elements in queue
      while (!queue.isEmpty()) {
         System.out.printf("%.1f ", queue.peek()); // view top element
         queue.poll(); // remove top element
      }
   }
}
```

```
Polling from queue: 3.2 5.4 9.8
```

https://www.ssa.gov/employer/randomization.html

```java
// Fig. 12.13: SetTest.java
// HashSet used to remove duplicate values from an array of strings.
import java.util.List;
import java.util.Arrays;
import java.util.HashSet;
import java.util.Set;
import java.util.Collection;

public class SetTest {
   public static void main(String[] args) {
      // create and display a List<String>
      String[] colors = {"red", "white", "blue", "green", "gray",
         "orange", "tan", "white", "cyan", "peach", "gray", "orange"};
      List<String> list = Arrays.asList(colors);
      System.out.printf("List: %s%n", list);

      // eliminate duplicates then print the unique values
      printNonDuplicates(list);
   }

   // create a Set from a Collection to eliminate duplicates
   private static void printNonDuplicates(Collection<String> values) {
      // create a HashSet
      Set<String> set = new HashSet<>(values);

      System.out.printf("%nNonduplicates are: ");

      for (String value : set) {
         System.out.printf("%s ", value);
      }

      System.out.println();
   }
}
```

```
List: [red, white, blue, green, gray, orange, tan, white, cyan, peach, gray,
orange]

Nonduplicates are: red orange tan green gray white blue peach cyan
```

```java
// Fig. 12.14: SortedSetTest.java
// Using SortedSets and TreeSets.
import java.util.Arrays;
import java.util.SortedSet;
import java.util.TreeSet;

public class SortedSetTest {
   public static void main(String[] args) {
      // create TreeSet from array colors
      String[] colors = {"yellow", "green", "black", "tan", "grey",
         "white", "orange", "red", "green"};
      SortedSet<String> tree = new TreeSet<>(Arrays.asList(colors));

      System.out.print("sorted set: ");
      printSet(tree);

      // get headSet based on "orange"
      System.out.print("headSet (\"orange\"):  ");
      printSet(tree.headSet("orange"));

      // get tailSet based on "orange"
      System.out.print("tailSet (\"orange\"):  ");
      printSet(tree.tailSet("orange"));

      // get first and last elements
      System.out.printf("first: %s%n", tree.first());
      System.out.printf("last : %s%n", tree.last());
   }

   // output SortedSet using enhanced for statement
   private static void printSet(SortedSet<String> set) {
      for (String s : set) {
         System.out.printf("%s ", s);
      }

      System.out.println();
   }
}
```

```
sorted set: black green grey orange red tan white yellow
headSet ("orange"):  black green grey
tailSet ("orange"):  orange red tan white yellow
first: black
last : yellow
```

```java
// Fig. 12.15: WordTypeCount.java
// Program counts the number of occurrences of each word in a String.
import java.util.Map;
import java.util.HashMap;
import java.util.Set;
import java.util.TreeSet;
import java.util.Scanner;

public class WordTypeCount {
   public static void main(String[] args) {
      // create HashMap to store String keys and Integer counts
      Map<String, Integer> myMap = new HashMap<>();

      createMap(myMap); // create map based on user input
      displayMap(myMap); // display map content
   }

   // create map from user input
   private static void createMap(Map<String, Integer> map) {
      var scanner = new Scanner(System.in); // create scanner
      System.out.println("Enter a string:"); // prompt for user input
      String input = scanner.nextLine();

      // tokenize the input
      String[] tokens = input.split(" ");

      // processing input text
      for (String token : tokens) {
         String word = token.toLowerCase(); // get lowercase word

         // if the map contains the word
         if (map.containsKey(word)) { // is word in map?
            int count = map.get(word); // get current count
            map.put(word, count + 1); // increment count
         }
         else {
            map.put(word, 1); // add new word with a count of 1 to map
         }
      }
   }


   // display map content
   private static void displayMap(Map<String, Integer> map) {
      Set<String> keys = map.keySet(); // get a Set of the keys

      // sort keys
      var sortedKeys = new TreeSet<String>(keys);

      System.out.printf("%nMap contains:%nKey\t\tValue%n");

      // generate output for each key in map
      for (String key : sortedKeys) {
         System.out.printf("%-10s%10s%n", key, map.get(key));
      }

      System.out.printf(
         "%nsize: %d%nisEmpty: %b%n", map.size(), map.isEmpty());
   }
}
```

```
Enter a string:
this is a sample sentence with several words this is another sample
sentence with several different words

Map contains:
Key             Value
a                   1
another             1
different           1
is                  2
sample              2
sentence            2
several             2
this                2
with                2
words               2

size: 10
isEmpty: false
```

```java
// Fig. 12.16: FactoryMethods.java
// Convenience factory methods for creating immutable collections.
import java.util.List;
import java.util.Map;
import java.util.Set;

public class FactoryMethods {
   public static void main(String[] args) {
      // create a List
      List<String> colorList = List.of("red", "orange", "yellow",
         "green", "blue", "indigo", "violet");
      System.out.printf("colorList: %s%n%n", colorList);

      // create a Set
      Set<String> colorSet = Set.of("red", "orange", "yellow",
         "green", "blue", "indigo", "violet");
      System.out.printf("colorSet: %s%n%n", colorSet);

      // create a Map using method "of"
      Map<String, Integer> dayMap = Map.of("Monday", 1, "Tuesday", 2,
         "Wednesday", 3, "Thursday", 4, "Friday", 5, "Saturday", 6,
         "Sunday", 7);
      System.out.printf("dayMap: %s%n%n", dayMap);

      // create a Map using method "ofEntries" for more than 10 pairs
      Map<String, Integer> daysPerMonthMap = Map.ofEntries(
         Map.entry("January", 31),
         Map.entry("February", 28),
         Map.entry("March", 31),
         Map.entry("April", 30),
         Map.entry("May", 31),
         Map.entry("June", 30),
         Map.entry("July", 31),
         Map.entry("August", 31),
         Map.entry("September", 30),
         Map.entry("October", 31),
         Map.entry("November", 30),
         Map.entry("December", 31)
      );
      System.out.printf("monthMap: %s%n", daysPerMonthMap);
   }
}
```

```
colorList: [red, orange, yellow, green, blue, indigo, violet]

colorSet: [yellow, green, red, blue, violet, indigo, orange]

dayMap: {Tuesday=2, Wednesday=3, Friday=5, Thursday=4, Saturday=6, Monday=1,
Sunday=7}

monthMap: {April=30, February=28, September=30, July=31, October=31,
November=30, December=31, March=31, January=31, June=30, May=31, August=31}
```

```
colorList: [red, orange, yellow, green, blue, indigo, violet]

colorSet: [violet, yellow, orange, green, blue, red, indigo]

dayMap: {Saturday=6, Tuesday=2, Wednesday=3, Sunday=7, Monday=1, Thursday=4,
Friday=5}

monthMap: {February=28, August=31, July=31, November=30, April=30, May=31,
December=31, September=30, January=31, March=31, June=30, October=31}
```

https://docs.oracle.com/javase/tutorial/collections/index.html

https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/util/doc-files/coll-index.html.

```java
// Fig. 13.1: OverloadedMethods.java
// Printing array elements using overloaded methods.

public class OverloadedMethods {
   public static void main(String[] args) {
      // create arrays of Integers and Doubles
      Integer[] integerArray = {1, 2, 3, 4, 5, 6};
      Double[] doubleArray = {1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7};

      System.out.printf("Array integerArray contains: ");
      printArray(integerArray); // pass an Integer array
      System.out.printf("Array doubleArray contains: ");
      printArray(doubleArray); // pass a Double array
   }

   // method printArray to print Integer array
   public static void printArray(Integer[] inputArray) {
      // display array elements
      for (Integer element : inputArray) {
         System.out.printf("%s ", element);
      }

      System.out.println();
   }

   // method printArray to print Double array
   public static void printArray(Double[] inputArray) {
      // display array elements
      for (Double element : inputArray) {
         System.out.printf("%s ", element);
      }

      System.out.println();
   }
}
```

```
Array integerArray contains: 1 2 3 4 5 6
Array doubleArray contains: 1.1 2.2 3.3 4.4 5.5 6.6 7.7
```

```java
public static void printArray(T[] inputArray) {
   // display array elements
   for (T element : inputArray)  {
      System.out.printf("%s ", element);
   }

   System.out.println();
}
```

```java
// Fig. 13.2: GenericMethodTest.java
// Printing array elements using generic method printArray.

public class GenericMethodTest {
   public static void main(String[] args) {
      // create arrays of Integer, Double and Character
      Integer[] integerArray = {1, 2, 3, 4, 5, 6};
      Double[] doubleArray = {1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7};

      System.out.printf("Array integerArray contains: ");
      printArray(integerArray); // pass an Integer array
      System.out.printf("Array doubleArray contains: ");
      printArray(doubleArray); // pass a Double array
   }

   // generic method printArray
   public static <T> void printArray(T[] inputArray) {
      // display array elements
      for (T element : inputArray) {
         System.out.printf("%s ", element);
      }

      System.out.println();
   }
}
```

```
Array integerArray contains: 1 2 3 4 5 6
Array doubleArray contains: 1.1 2.2 3.3 4.4 5.5 6.6 7.7
```

public static <T> T maximum(T value1, T value2)

https://docs.oracle.com/javase/specs/jls/se22/html/jls-15.html#jls-15.12

```java
public static void printArray(Object[] inputArray) {
   // display array elements
   for (Object element : inputArray) {
      System.out.printf("%s ", element);
   }

   System.out.println();
}
```

```java
// Fig. 13.3: MaximumTest.java
// Generic method maximum returns the largest of three objects.

public class MaximumTest {
   public static void main(String[] args) {
      System.out.printf("Maximum of %d, %d and %d is %d%n", 3, 4, 5,
         maximum(3, 4, 5));
      System.out.printf("Maximum of %.1f, %.1f and %.1f is %.1f%n",
         6.6, 8.8, 7.7, maximum(6.6, 8.8, 7.7));
      System.out.printf("Maximum of %s, %s and %s is %s%n", "pear",
         "apple", "orange", maximum("pear", "apple", "orange"));
   }

   // determines the largest of three objects
   public static <T extends Comparable<T>> T maximum(T x, T y, T z) {
      T max = x; // assume x is initially the largest

      if (y.compareTo(max) > 0) {
         max = y; // y is the largest so far
      }

      if (z.compareTo(max) > 0) {
         max = z; // z is the largest
      }

      return max; // returns the largest object
   }
}
```

```
Maximum of 3, 4 and 5 is 5
Maximum of 6.6, 8.8 and 7.7 is 8.8
Maximum of pear, apple and orange is pear
```

```
MaximumTest.java:18: error: cannot find symbol
      if (y.compareTo(max) > 0) {
            ^
  symbol:   method compareTo(T)
  location: variable y of type T
  where T is a type-variable:
    T extends Object declared in method <T>maximum(T,T,T)
```

```java
public static Comparable maximum(Comparable x, Comparable y,
   Comparable z) {

   Comparable max = x; // assume x is initially the largest

   if (y.compareTo(max) > 0) {
      max = y; // y is the largest so far
   }

   if (z.compareTo(max) > 0) {
      max = z; // z is the largest
   }

   return max; // returns the largest object
}
```

```java
(Integer) maximum(3, 4, 5)
(Double) maximum(6.6, 8.8, 7.7)
(String) maximum("pear", "apple", "orange")
```

```java
// Fig. 13.4: Stack.java
// Stack generic class declaration.
import java.util.ArrayList;
import java.util.NoSuchElementException;

public class Stack<E> {
   private final ArrayList<E> elements; // ArrayList stores stack elements

   // no-argument constructor creates a stack of the default size
   public Stack() {
      this(10); // default stack size
   }

   // constructor creates a stack of the specified number of elements
   public Stack(int capacity) {
      int initCapacity = capacity > 0 ? capacity : 10; // validate
      elements = new ArrayList<E>(initCapacity); // create ArrayList
   }

   // push element onto stack
   public void push(E pushValue) {
      elements.addLast(pushValue); // place pushValue on Stack
   }

   // return the top element if not empty; else throw exception
   public E pop() {
      if (elements.isEmpty()) { // if stack is empty
         throw new NoSuchElementException("Stack is empty, cannot pop");
      }

      // remove and return top element of Stack
      return elements.removeLast();
   }
}
```

```java
// Fig. 13.5: StackTest.java
// Stack generic class test program.
import java.util.NoSuchElementException;

public class StackTest {
   public static void main(String[] args) {
      double[] doubleElements = {1.1, 2.2, 3.3, 4.4, 5.5};
      int[] integerElements = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

      // Create a Stack<Double> and a Stack<Integer>
      var doubleStack = new Stack<Double>(5);
      var integerStack = new Stack<Integer>();

      // push elements of doubleElements onto doubleStack
      testPushDouble(doubleStack, doubleElements);
      testPopDouble(doubleStack); // pop from doubleStack

      // push elements of integerElements onto integerStack
      testPushInteger(integerStack, integerElements);
      testPopInteger(integerStack); // pop from integerStack
   }

   // test push method with double stack
   private static void testPushDouble(
      Stack<Double> stack, double[] values) {
      System.out.printf("%nPushing elements onto doubleStack%n   ");
```

```
27
28          // push elements to Stack
29          for (double value : values) {
30             System.out.printf("%.1f ", value);
31             stack.push(value); // push onto doubleStack
32          }
33       }
34
35       // test pop method with double stack
36       private static void testPopDouble(Stack<Double> stack) {
37          // pop elements from stack
38          try {
39             System.out.printf("%nPopping elements from doubleStack%n   ");
40             double popValue; // store element removed from stack
41
42             // remove all elements from Stack
43             while (true) {
44                popValue = stack.pop(); // pop from doubleStack
45                System.out.printf("%.1f ", popValue);
46             }
47          }
48          catch(NoSuchElementException noSuchElementException) {
49             System.err.println();
50             noSuchElementException.printStackTrace();
51          }
52       }


53
54       // test push method with integer stack
55       private static void testPushInteger(
56          Stack<Integer> stack, int[] values) {
57          System.out.printf("%nPushing elements onto integerStack%n   ");
58
59          // push elements to Stack
60          for (int value : values) {
61             System.out.printf("%d ", value);
62             stack.push(value); // push onto integerStack
63          }
64       }
65
66       // test pop method with integer stack
67       private static void testPopInteger(Stack<Integer> stack) {
68          // pop elements from stack
69          try {
70             System.out.printf("%nPopping elements from integerStack%n   ");
71             int popValue; // store element removed from stack
72
73             // remove all elements from Stack
74             while (true) {
75                popValue = stack.pop(); // pop from intStack
76                System.out.printf("%d ", popValue);


77             }
78          }
79          catch(NoSuchElementException noSuchElementException) {
80             System.err.println();
81             noSuchElementException.printStackTrace();
82          }
83       }
84    }
```

```
Pushing elements onto doubleStack
   1.1 2.2 3.3 4.4 5.5
Popping elements from doubleStack
   5.5 4.4 3.3 2.2 1.1
java.util.NoSuchElementException: Stack is empty, cannot pop
        at Stack.pop(Stack.java:28)
        at StackTest.testPopDouble(StackTest.java:44)
        at StackTest.main(StackTest.java:16)

Pushing elements onto integerStack
   1 2 3 4 5 6 7 8 9 10
Popping elements from integerStack
   10 9 8 7 6 5 4 3 2 1
java.util.NoSuchElementException: Stack is empty, cannot pop
        at Stack.pop(Stack.java:28)
        at StackTest.testPopInteger(StackTest.java:75)
        at StackTest.main(StackTest.java:20)
```

```
popValue = (Double) stack.pop();

popValue = (Integer) stack.pop();
```

```
1  // Fig. 13.6: StackTest2.java
2  // Passing generic Stack objects to generic methods.
3  import java.util.NoSuchElementException;
4
5  public class StackTest2 {
6     public static void main(String[] args) {
7        Double[] doubleElements = {1.1, 2.2, 3.3, 4.4, 5.5};
8        Integer[] integerElements = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
9
10       // Create a Stack<Double> and a Stack<Integer>
11       Stack<Double> doubleStack = new Stack<>(5);
12       Stack<Integer> integerStack = new Stack<>();
13
14       // push elements of doubleElements onto doubleStack
15       testPush("doubleStack", doubleStack, doubleElements);
16       testPop("doubleStack", doubleStack); // pop from doubleStack
17
18       // push elements of integerElements onto integerStack
19       testPush("integerStack", integerStack, integerElements);
20       testPop("integerStack", integerStack); // pop from integerStack
21    }
22
23    // generic method testPush pushes elements onto a Stack
24    public static <E> void testPush(String name, Stack<E> stack,
25       E[] elements) {
26       System.out.printf("%nPushing elements onto %s%n   ", name);
27
28       // push elements onto Stack
29       for (E element : elements) {
30          System.out.printf("%s ", element);
31          stack.push(element); // push element onto stack
32       }
33    }
34
35    // generic method testPop pops elements from a Stack
36    public static <E> void testPop(String name, Stack<E> stack) {
37       // pop elements from stack
38       try {
39          System.out.printf("%nPopping elements from %s%n   ", name);
40          E popValue; // store element removed from stack
41
42          // remove all elements from Stack
43          while (true) {
44             popValue = stack.pop();
45             System.out.printf("%s ", popValue);
46          }
47       }
48       catch(NoSuchElementException noSuchElementException) {
49          System.out.println();
50          noSuchElementException.printStackTrace();
51       }
52    }
53 }
```

```
Pushing elements onto doubleStack
   1.1 2.2 3.3 4.4 5.5
Popping elements from doubleStack
   5.5 4.4 3.3 2.2 1.1
java.util.NoSuchElementException: Stack is empty, cannot pop
        at Stack.pop(Stack.java:28)
        at StackTest2.testPop(StackTest2.java:44)
        at StackTest2.main(StackTest2.java:16)

Pushing elements onto integerStack
   1 2 3 4 5 6 7 8 9 10
Popping elements from integerStack
   10 9 8 7 6 5 4 3 2 1
java.util.NoSuchElementException: Stack is empty, cannot pop
        at Stack.pop(Stack.java:28)
        at StackTest2.testPop(StackTest2.java:44)
        at StackTest2.main(StackTest2.java:20)
```

```java
// Fig. 13.7: TotalNumbers.java
// Totaling the numbers in a List<Number>.
import java.util.Arrays;
import java.util.List;

public class TotalNumbers {
   public static void main(String[] args) {
      // create List<Number> and add Integers and Doubles to it
      Number[] numbers = {1, 2.4, 3, 4.1};
      List<Number> numberList = Arrays.asList(numbers);

      // display numbersList
      System.out.printf("numberList contains: %s%n", numberList);

      // total numbersList's elements then display the result
      System.out.printf("Total of the elements in numberList: %.1f%n",
         sum(numberList));
   }

   // calculate total of List elements
   public static double sum(List<Number> list) {
      double total = 0; // initialize total

      // calculate sum
      for (Number element : list) {
         total += element.doubleValue();
      }

      return total;
   }
}
```

```
numberList contains: [1, 2.4, 3, 4.1]
Total of the elements in numberList: 10.5
```

```
TotalNumbersErrors.java:17: error: method sum in class
TotalNumbersErrors cannot be applied to given types;
         sum(numberList));
         ^
  required: List<Number>
  found:    List<Integer>
  reason: argument mismatch; List<Integer> cannot be converted to
          List<Number>
1 error
error: compilation failed
```

```java
// Fig. 13.8: WildcardTest.java
// Wildcard test program.
import java.util.Arrays;
import java.util.List;

public class WildcardTest {
   public static void main(String[] args) {
      // create and initialize a List<Integer>
      Integer[] integers = {1, 2, 3, 4, 5};
      List<Integer> integerList = Arrays.asList(integers);

      System.out.printf("integerList contains: %s%n", integerList);
      System.out.printf("Total of the elements in integerList: %.0f%n%n",
         sum(integerList));

      // create and initialize a List<Double>
      Double[] doubles = {1.1, 3.3, 5.5};
      List<Double> doubleList =  Arrays.asList(doubles);

      System.out.printf("doubleList contains: %s%n", doubleList);
      System.out.printf("Total of the elements in doubleList: %.1f%n%n",
         sum(doubleList));

      // create and initialize a List<Number> with mixed numeric types
      Number[] numbers = {1, 2.4, 3, 4.1};
      List<Number> numberList = Arrays.asList(numbers);
```

```
27
28          System.out.printf("numberList contains: %s%n", numberList);
29          System.out.printf("Total of the elements in numberList: %.1f%n",
30             sum(numberList));
31       }
32
33       // total the elements; using a wildcard in the List parameter
34       public static double sum(List<? extends Number> list) {
35          double total = 0; // initialize total
36
37          // calculate sum
38          for (Number element : list) {
39             total += element.doubleValue();
40          }
41
42          return total;
43       }
44    }
```

```
integerList contains: [1, 2, 3, 4, 5]
Total of the elements in integerList: 15

doubleList contains: [1.1, 3.3, 5.5]
Total of the elements in doubleList: 9.9

numberList contains: [1, 2.4, 3, 4.1]
Total of the elements in numberList: 10.5
```

```
public static <T extends Number> double sum(List<T> list)
```

```
int total = 0;

for (int number = 1; number <= 10; ++number) {
   total += number;
}
```

```
1  // Fig. 14.1: StreamReduce.java
2  // Sum the integers from 1 through 10 with IntStream.
3  import java.util.stream.IntStream;
4
5  public class StreamReduce {
6     public static void main(String[] args) {
7        System.out.printf("Sum of 1 through 10 is: %d%n",
8           IntStream.rangeClosed(1, 10)
9                    .sum());
10    }
11 }
```

```
Sum of 1 through 10 is: 55
```

```
int total = 0;

for (int number = 2; number <= 20; number += 2) {
   total += number;
}
```

```
1  // Fig. 14.2: StreamMapReduce.java
2  // Sum the even integers from 2 through 20 with IntStream.
3  import java.util.stream.IntStream;
4
5  public class StreamMapReduce {
6     public static void main(String[] args) {
7        System.out.printf("Sum of the even ints from 2 through 20 is: %d%n",
8           IntStream.rangeClosed(1, 10)              // 1...10
9                    .map((int x) -> {return x * 2;}) // multiply by 2
10                   .sum());                         // sum
11    }
12 }
```

```
Sum of the even ints from 2 through 20 is: 110
```

```
() -> System.out.println("Welcome to lambdas!")
```

```
int total = 0;

for (int x = 1; x <= 10; ++x) {
   if (x % 2 == 0) { // if x is even
      total += x * 2;
   }
}
```

```java
// Fig. 14.3: StreamFilterMapReduce.java
// Double the even ints from 2 through 10 then sum them with IntStream.
import java.util.stream.IntStream;

public class StreamFilterMapReduce {
   public static void main(String[] args) {
      System.out.printf(
         "Sum of the doubles of the even ints from 2 through 10 is: %d%n",
         IntStream.rangeClosed(1, 10)
                  .filter(x -> x % 2 == 0)
                  .map(x -> x * 2)
                  .sum());
   }
}
```

```
Sum of the doubles of the even ints from 2 through 10 is: 60
```

For each element
    If the element is an even integer
        Multiply the element by 2 and add the result to the total

```java
IntStream.rangeClosed(1, 10)
         .filter(
            x -> {
               System.out.printf("%nfilter: %d%n", x);
               return x % 2 == 0;
            })
         .map(
            x -> {
               System.out.println("map: " + x);
               return x * 2;
            })
         .sum()
```

```
filter: 1 // odd so no map step is performed for this element

filter: 2 // even so a map step is performed next
map: 2

filter: 3 // odd so no map step is performed for this element

filter: 4 // even so a map step is performed next
map: 4

filter: 5 // odd so no map step is performed for this element

filter: 6 // even so a map step is performed next
map: 6

filter: 7 // odd so no map step is performed for this element

filter: 8 // even so a map step is performed next
map: 8

filter: 9 // odd so no map step is performed for this element

filter: 10 // even so a map step is performed next
map: 10
```

```
1  // Fig. 14.4: RandomIntegers.java
2  // Shifted and scaled random integers.
3  import java.util.random.RandomGenerator;
4  import java.util.stream.Collectors;
5
6  public class RandomIntegers {
7     public static void main(String[] args) {
8        var random = RandomGenerator.getDefault();
9
10       // display 10 random integers on separate lines
11       System.out.println("Random numbers on separate lines:");
12       random.ints(10, 1, 7)
13             .forEach(System.out::println);
14
15       // display 10 random integers on the same line
16       String numbers =
17          random.ints(10, 1, 7)
18                .mapToObj(String::valueOf)
19                .collect(Collectors.joining(" "));
20       System.out.printf("%nRandom numbers on one line: %s%n", numbers);
21
22    }
23 }
```

```
Random numbers on separate lines:
4
3
4
5
1
5
5
3
6
5

Random numbers on one line: 4 6 2 5 6 4 3 2 4 1
```

objectName::*instanceMethodName*

```
for (int i = 1; i <= 10; ++i) {
   System.out.println(random.nextInt(1, 7));
}
```

```
String numbers =
   random.ints(10, 1, 7)
         .mapToObj(String::valueOf)
         .collect(Collectors.joining(" "));
```

```
1  // Fig. 14.5: IntStreamOperations.java
2  // Demonstrating IntStream operations.
3  import java.util.stream.Collectors;
4  import java.util.stream.IntStream;
5
6  public class IntStreamOperations {
7     public static void main(String[] args) {
8        int[] values = {3, 10, 6, 1, 4, 8, 2, 5, 9, 7};
9
```

```
10         // display original values
11         System.out.print("Original values: ");
12         System.out.println(
13            IntStream.of(values)
14                     .mapToObj(String::valueOf)
15                     .collect(Collectors.joining(" ")));
16
```

```
Original values: 3 10 6 1 4 8 2 5 9 7
```

```
17         // count, min, max, sum and average of the values
18         System.out.printf("%nCount: %d%n", IntStream.of(values).count());
19         System.out.printf("Min: %d%n",
20            IntStream.of(values).min().getAsInt());
21         System.out.printf("Max: %d%n",
22            IntStream.of(values).max().getAsInt());
23         System.out.printf("Sum: %d%n", IntStream.of(values).sum());
24         System.out.printf("Average: %.2f%n",
25            IntStream.of(values).average().getAsDouble());
26
```

```
Count: 10
Min: 1
Max: 10
Sum: 55
Average: 5.50
```

```
IntStream.of(values).min().orElse(-1)
```

```
System.out.println(IntStream.of(values).summaryStatistics());
```

```
IntSummaryStatistics{count=10, sum=55, min=1, average=5.500000,
max=10}
```

```
27         // sum of values with reduce method
28         System.out.printf("%nSum via reduce method: %d%n",
29            IntStream.of(values)
30                     .reduce(0, (x, y) -> x + y));
31
```

```
Sum via reduce method: 55
```

```
IntStream.of(values)
         .reduce(0, (x, y) -> x + y)
```

```
32         // product of values with reduce method
33         System.out.printf("Product via reduce method: %d%n",
34            IntStream.of(values)
35                     .reduce((x, y) -> x * y).getAsInt());
36
```

```
Product via reduce method: 3628800
```

```
IntStream.of(values)
         .reduce(1, (x, y) -> x * y)
```

```
37         // sum of squares of values with map and sum methods
38         System.out.printf("Sum of squares via map and sum: %d%n%n",
39            IntStream.of(values)
40                     .map(x -> x * x)
41                     .sum());
42
```

```
Sum of squares via map and sum: 385
```

```
43         // displaying the elements in sorted order
44         System.out.printf("Values displayed in sorted order: %s%n",
45            IntStream.of(values)
46                     .sorted()
47                     .mapToObj(String::valueOf)
48                     .collect(Collectors.joining(" ")));
49      }
50   }
```

```
Values displayed in sorted order: 1 2 3 4 5 6 7 8 9 10
```

```
IntStream.of(values)
         .sorted()
         .mapToObj(String::valueOf)
         .collect(Collectors.joining(" ")));

IntStream.rangeClosed(1, 10)
         .map((int x) -> {return x * 2;})
         .sum()

IntStream.rangeClosed(1, 10)
         .filter(x -> x % 2 == 0)
         .map(x -> x * 2)
         .sum()
```

```
1  // Fig. 14.6: ArraysAndStreams.java
2  // Demonstrating lambdas and streams with an array of Integers.
3  import java.util.Arrays;
4  import java.util.List;
5  import java.util.stream.Collectors;
6
7  public class ArraysAndStreams {
8     public static void main(String[] args) {
9        Integer[] values = {2, 9, 5, 0, 3, 7, 1, 4, 8, 6};
10
11       // display original values
12       System.out.printf("Original values: %s%n", Arrays.asList(values));
13
```

```
Original values: [2, 9, 5, 0, 3, 7, 1, 4, 8, 6]
```

```
14       // sort values in ascending order with streams
15       System.out.printf("Sorted values: %s%n",
16          Arrays.stream(values)
17                .sorted()
18                .collect(Collectors.toList()));
19
```

```
Sorted values: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
Arrays.stream(values)
      .sorted()
      .collect(Collectors.toList())
```

https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/util/stream/Collectors.html

```
20         // values greater than 4
21         List<Integer> greaterThan4 =
22            Arrays.stream(values)
23                  .filter(value -> value > 4)
24                  .toList());
25         System.out.printf("Values greater than 4: %s%n", greaterThan4);
26
```

```
Values greater than 4: [9, 5, 7, 8, 6]
```

```
27         // filter values greater than 4 then sort the results
28         System.out.printf("Sorted values greater than 4: %s%n",
29            Arrays.stream(values)
30                  .filter(value -> value > 4)
31                  .sorted()
32                  .toList());
33
```

```
Sorted values greater than 4: [5, 6, 7, 8, 9]
```

```
34         // greaterThan4 List sorted with streams
35         System.out.printf(
36            "Values greater than 4 (ascending with streams): %s%n",
37            greaterThan4.stream()
38                  .sorted()
39                  .toList());
40      }
41   }
```

```
Values greater than 4 (ascending with streams): [5, 6, 7, 8, 9]
```

```
1  // Fig. 14.7: ArraysAndStreams2.java
2  // Demonstrating lambdas and streams with an array of Strings.
3  import java.util.Arrays;
4
5  public class ArraysAndStreams2 {
6     public static void main(String[] args) {
7        String[] strings =
8           {"Red", "orange", "Yellow", "green", "Blue", "indigo", "Violet"};
9
10       // display original strings
11       System.out.printf("Original strings: %s%n", Arrays.asList(strings));
12
```

```
Original strings: [Red, orange, Yellow, green, Blue, indigo, Violet]
```

```
13       // strings in uppercase
14       System.out.printf("strings in uppercase: %s%n",
15          Arrays.stream(strings)
16                .map(String::toUpperCase)
17                .toList());
18
```

```
strings in uppercase: [RED, ORANGE, YELLOW, GREEN, BLUE, INDIGO, VIOLET]
```

```
        // strings less than "n" (case insensitive) sorted ascending
        System.out.printf("strings less than n sorted ascending: %s%n",
           Arrays.stream(strings)
                 .filter(s -> s.compareToIgnoreCase("n") < 0)
                 .sorted(String.CASE_INSENSITIVE_ORDER)
                 .toList());

```

```
strings less than n sorted ascending: [Blue, green, indigo]
```

```
        // strings less than "n" (case insensitive) sorted descending
        System.out.printf("strings less than n sorted descending: %s%n",
           Arrays.stream(strings)
                 .filter(s -> s.compareToIgnoreCase("n") < 0)
                 .sorted(String.CASE_INSENSITIVE_ORDER.reversed())
                 .toList());
     }
  }
```

```
strings less than n sorted descending: [indigo, green, Blue]
```

```
// Fig. 14.8: Employee.java
// Employee class.
public record Employee(
   String firstName, String lastName, double salary, String department) {
   // return Employee's first and last name combined
   public String getName() {
      return String.format("%s %s", firstName(), lastName());
   }

   // return a String containing the Employee's information
   @Override
   public String toString() {
      return String.format("%-8s %-8s %8.2f   %s",
         firstName(), lastName(), salary(), department());
   }
}
```

```
// Fig. 14.9: ProcessingEmployees.java
// Processing streams of Employee objects.
import java.util.Comparator;
import java.util.List;
import java.util.Map;
import java.util.function.Function;
import java.util.function.Predicate;
import java.util.stream.Collectors;

public class ProcessingEmployees {
   public static void main(String[] args) {

      // initialize List<Employee>
      List<Employee> list = List.of(
         new Employee("James", "Roy", 5000, "IT"),
         new Employee("Zaynab", "Hanna", 7600, "IT"),
         new Employee("Zoe", "Lopez", 3587.5, "Sales"),
         new Employee("Mateo", "Lopez", 4700.77, "Marketing"),
         new Employee("Diego", "Lopez", 3200, "IT"),
         new Employee("Emma", "Bernard", 6200, "Sales"),
         new Employee("Xinyi", "Chen", 4236.4, "Marketing"));

```

```
23          // display all Employees
24          System.out.println("Complete Employee list:");
25          list.stream().forEach(System.out::println);
26
```

```
Complete Employee list:
James    Roy       5000.00   IT
Zaynab   Hanna     7600.00   IT
Zoe      Lopez     3587.50   Sales
Mateo    Lopez     4700.77   Marketing
Diego    Lopez     3200.00   IT
Emma     Bernard   2200.00   Sales
Xinyi    Chen      4236.40   Marketing
```

```
27          // Predicate that returns true for salaries in the range $4000-$6000
28          Predicate<Employee> salaryInRange4to6k =
29             e -> (e.salary() >= 4000 && e.salary() <= 6000);
30
31          // Display Employees with salaries in the range $4000-$6000
32          // sorted into ascending order by salary
33          System.out.printf(
34             "%nEmployees earning $4000-$6000 per month sorted by salary:%n");
35          list.stream()
36              .filter(salaryInRange4to6k)
37              .sorted(Comparator.comparing(Employee::salary))
38              .forEach(System.out::println);
39
```

```
Employees earning $4000-$6000 per month sorted by salary:
Xinyi    Chen      4236.40   Marketing
Mateo    Lopez     4700.77   Marketing
James    Roy       5000.00   IT
```

```
40          // Display first Employee with salary in the range $4000-$6000
41          System.out.printf("%nFirst employee who earns $4000-$6000:%n%s%n",
42             list.stream()
43                 .filter(salaryInRange4to6k)
44                 .findFirst()
45                 .get());
46
```

```
First employee who earns $4000-$6000:
James    Roy       5000.00   IT
```

```
47          // Functions for getting first and last names from an Employee
48          Function<Employee, String> byFirstName = Employee::firstName;
49          Function<Employee, String> byLastName = Employee::lastName;
50
51          // Comparator for comparing Employees by first name then last name
52          Comparator<Employee> lastThenFirst =
53             Comparator.comparing(byLastName).thenComparing(byFirstName);
54
55          // sort employees by last name, then first name
56          System.out.printf(
57             "%nEmployees in ascending order by last name then first:%n");
58          list.stream()
59              .sorted(lastThenFirst)
60              .forEach(System.out::println);
61
62          // sort employees in descending order by last name, then first name
63          System.out.printf(
64             "%nEmployees in descending order by last name then first:%n");
65          list.stream()
66              .sorted(lastThenFirst.reversed())
67              .forEach(System.out::println);
68
```

```
Employees in ascending order by last name then first:
Emma     Bernard   6200.00   Sales
Xinyi    Chen      4236.40   Marketing
Zaynab   Hanna     7600.00   IT
Diego    Lopez     3200.00   IT
Mateo    Lopez     4700.77   Marketing
Zoe      Lopez     3587.50   Sales
James    Roy       5000.00   IT

Employees in descending order by last name then first:
James    Roy       5000.00   IT
Zoe      Lopez     3587.50   Sales
Mateo    Lopez     4700.77   Marketing
Diego    Lopez     3200.00   IT
Zaynab   Hanna     7600.00   IT
Xinyi    Chen      4236.40   Marketing
Emma     Bernard   6200.00   Sales
```

```
Comparator<Employee> lastThenFirst =
   Comparator.comparing(Employee::lastName)
             .thenComparing(Employee::firstName);
```

```
IntPredicate even = value -> value % 2 == 0;
IntPredicate greaterThan5 = value -> value > 5;
```

```
69          // display unique employee last names sorted
70          System.out.printf("%nUnique employee last names:%n");
71          list.stream()
72              .map(Employee::lastName)
73              .distinct()
74              .sorted()
75              .forEach(System.out::println);
76
77          // display only first and last names
78          System.out.printf(
79             "%nEmployee names in order by last name then first name:%n");
80          list.stream()
81              .sorted(lastThenFirst)
82              .map(Employee::getName)
83              .forEach(System.out::println);
84
```

```
Unique employee last names:
Bernard
Chen
Hanna
Lopez
Roy

Employee names in order by last name then first name:
Emma Bernard
Xinyi Chen
Zaynab Hanna
Diego Lopez
Mateo Lopez
Zoe Lopez
James Roy
```

```
85          // group Employees by department
86          System.out.printf("%nEmployees by department:%n");
87          Map<String, List<Employee>> groupedByDepartment =
88             list.stream()
89                 .collect(Collectors.groupingBy(Employee::department));
90          groupedByDepartment.forEach(
91             (department, employeesInDepartment) -> {
92                System.out.printf("%n%s%n", department);
93                employeesInDepartment.forEach(
94                   employee -> System.out.printf("   %s%n", employee));
95             }
96          );
97
```

```
Employees by department:

Sales
   Zoe      Lopez     3587.50   Sales
   Emma     Bernard   6200.00   Sales

IT
   James    Roy       5000.00   IT
   Zaynab   Hanna     7600.00   IT
   Diego    Lopez     3200.00   IT

Marketing
   Mateo    Lopez     4700.77   Marketing
   Xinyi    Chen      4236.40   Marketing
```

```
98          // count number of Employees in each department
99          System.out.printf("%nCount of Employees by department:%n");
100         Map<String, Long> employeeCountByDepartment =
101            list.stream()
102                .collect(Collectors.groupingBy(Employee::department,
103                   Collectors.counting()));
104         employeeCountByDepartment.forEach(
105            (department, count) -> System.out.printf(
106               "%s has %d employee(s)%n", department, count));
107
```

```
Count of Employees by department:
Sales has 2 employee(s)
IT has 3 employee(s)
Marketing has 2 employee(s)
```

```
108         // sum of Employee salaries with DoubleStream sum method
109         System.out.printf(
110            "%nSum of Employees' salaries (via sum method): %.2f%n",
111            list.stream()
112                .mapToDouble(Employee::salary)
113                .sum());
114
115         // calculate sum of Employee salaries with Stream reduce method
116         System.out.printf(
117            "Sum of Employees' salaries (via reduce method): %.2f%n",
118            list.stream()
119                .mapToDouble(Employee::salary)
120                .reduce(0, (value1, value2) -> value1 + value2));
121
122         // average of Employee salaries with DoubleStream average method
123         System.out.printf("Average of Employees' salaries: %.2f%n",
124            list.stream()
125                .mapToDouble(Employee::salary)
126                .average()
127                .getAsDouble());
128      }
129   }
```

```
Sum of Employees' salaries (via sum method): 34524.67
Sum of Employees' salaries (via reduce method): 34524.67
Average of Employees' salaries: 4932.10
```

```
1   // Fig. 14.10: StreamOfLines.java
2   // Counting word occurrences in a text file.
3   import java.io.IOException;
4   import java.nio.file.Files;
5   import java.nio.file.Paths;
6   import java.util.Map;
7   import java.util.TreeMap;
8   import java.util.regex.Pattern;
9   import java.util.stream.Collectors;
10
11  public class StreamOfLines {
12     public static void main(String[] args) throws IOException {
13        // Regex that matches one or more consecutive whitespace characters
14        Pattern pattern = Pattern.compile("\\s+");
15
```

```
16          // count occurrences of each word in a Stream<String> sorted by word
17          Map<String, Long> wordCounts =
18             Files.lines(Paths.get("Chapter2Paragraph.txt"))
19                  .flatMap(line -> pattern.splitAsStream(line))
20                  .collect(Collectors.groupingBy(String::toLowerCase,
21                     TreeMap::new, Collectors.counting()));
22
```

```
        // display the words grouped by starting letter
        wordCounts.entrySet()
           .stream()
           .collect(
              Collectors.groupingBy(entry -> entry.getKey().charAt(0),
                 TreeMap::new, Collectors.toList()))
           .forEach((letter, wordList) -> {
              System.out.printf("%n%C%n", letter);
              wordList.stream().forEach(word -> System.out.printf(
                 "%13s: %d%n", word.getKey(), word.getValue()));
           });
     }
  }
```

```
// Fig. 14.12: RandomIntStream.java
// Rolling a die 60,000,000 times with streams
import java.util.random.RandomGenerator;
import java.util.function.Function;
import java.util.stream.Collectors;

public class RandomIntStream {
   public static void main(String[] args) {
      var random = RandomGenerator.getDefault();

      // roll a die 60,000,000 times and summarize the results
      System.out.printf("%-6s%s%n", "Face", "Frequency");
      random.ints(60_000_000, 1, 7)
            .boxed()
            .collect(Collectors.groupingBy(Function.identity(),
               Collectors.counting()))
            .forEach((face, frequency) ->
               System.out.printf("%-6d%d%n", face, frequency));
   }
}
```

```
Face   Frequency
1      9992993
2      10000363
3      10002272
4      10003810
5      10000321
6      10000241
```

Create an infinite stream representing all prime numbers
If the prime number is less than 10,000
Display the prime number

```
IntStream.iterate(1, x -> x + 1)
         .forEach(System.out::println);

IntStream.iterate(1, x -> x + 1)
         .limit(10)
         .forEach(System.out::println);

IntStream.iterate(1, x -> x + 1)
            .map(x -> x * x)
            .limit(10)
            .sum()
```

```
IntStream.generate(() -> random.nextInt())
         .limit(10)
         .forEach(System.out::println);
```

```
random.ints()
      .limit(10)
      .forEach(System.out::println);
```

```
javac --module-path pathToFX
   --add-modules javafx.controls,javafx.fxml *.java

java --module-path pathToFX
   --add-modules javafx.controls,javafx.fxml TipCalculator
```

https://openjfx.io/javadoc/23/javafx.graphics/javafx/scene/layout/GridPane.html

```java
// Fig. 15.1: TipCalculator.java
// Main application class that loads and displays the Tip Calculator's GUI
import javafx.application.Application;
import javafx.fxml.FXMLLoader;
import javafx.scene.Parent;
import javafx.scene.Scene;
import javafx.stage.Stage;

public class TipCalculator extends Application {
   @Override
   public void start(Stage stage) throws Exception {
      Parent root =
         FXMLLoader.load(getClass().getResource("TipCalculator.fxml"));

      var scene = new Scene(root); // attach scene graph to scene
      stage.setTitle("Tip Calculator"); // displayed in window's title bar
      stage.setScene(scene); // attach scene to stage
      stage.show(); // display the stage
   }

   public static void main(String[] args) {
      // create a TipCalculator object and call its start method
      launch(args);
   }
}
```

```java
// Fig. 15.2: TipCalculatorController.java
// Controller that handles calculateButton and tipPercentageSlider events
import java.math.BigDecimal;
import java.math.RoundingMode;
import java.text.NumberFormat;
import javafx.event.ActionEvent;
import javafx.fxml.FXML;
import javafx.scene.control.Label;
import javafx.scene.control.Slider;
import javafx.scene.control.TextField;

```

```
12  public class TipCalculatorController {
13     // formatters for currency and percentages
14     private static final NumberFormat currency =
15        NumberFormat.getCurrencyInstance();
16     private static final NumberFormat percent =
17        NumberFormat.getPercentInstance();
18
19     private BigDecimal tipPercentage = new BigDecimal("0.15"); // 15%
20
21     // GUI controls defined in FXML and used by the controller's code
22     @FXML
23     private TextField amountTextField;
24
25     @FXML
26     private Label tipPercentageLabel;
27
28     @FXML
29     private Slider tipPercentageSlider;
30
31     @FXML
32     private TextField tipTextField;
33
34     @FXML
35     private TextField totalTextField;
36
```

```
37     // calculates and displays the tip and total amounts
38     @FXML
39     private void calculateButtonPressed(ActionEvent event) {
40        try {
41           BigDecimal amount = new BigDecimal(amountTextField.getText());
42           BigDecimal tip = amount.multiply(tipPercentage);
43           BigDecimal total = amount.add(tip);
44
45           tipTextField.setText(currency.format(tip));
46           totalTextField.setText(currency.format(total));
47        }
48        catch (NumberFormatException ex) {
49           amountTextField.setText("Enter amount");
50           amountTextField.selectAll();
51           amountTextField.requestFocus();
52        }
53     }
54
```

```
55     // called by FXMLLoader to initialize the controller
56     @FXML
57     private void initialize() {
58        // 0-4 rounds down, 5-9 rounds up
59        currency.setRoundingMode(RoundingMode.HALF_EVEN);
60
61        // listener for changes to tipPercentageSlider's value
62        tipPercentageSlider.valueProperty().addListener(
63           (ov, oldValue, newValue) -> {
64              tipPercentage =
65                 BigDecimal.valueOf(newValue.intValue() / 100.0);
66              tipPercentageLabel.setText(percent.format(tipPercentage));
67           }
68        );
69     }
70  }
```

https://openjfx.io/javadoc/23/javafx.graphics/javafx/scene/Node.html

```
1  // Fig. 16.1: Painter.java
2  // Class that loads and displays the Painter's GUI.
3  import javafx.application.Application;
4  import javafx.fxml.FXMLLoader;
5  import javafx.scene.Parent;
6  import javafx.scene.Scene;
7  import javafx.stage.Stage;
8
9  public class Painter extends Application {
10    @Override
11    public void start(Stage stage) throws Exception {
12       Parent root =
13          FXMLLoader.load(getClass().getResource("Painter.fxml"));
14
15       Scene scene = new Scene(root);
16       stage.setTitle("Painter"); // displayed in window's title bar
17       stage.setScene(scene);
18       stage.show();
19    }
20
21    public static void main(String[] args) {
22       launch(args);
23    }
24 }
```

```
1  // Fig. 16.2: PainterController.java
2  // Controller for the Painter application
3  import javafx.event.ActionEvent;
4  import javafx.fxml.FXML;
5  import javafx.scene.control.RadioButton;
6  import javafx.scene.control.ToggleGroup;
7  import javafx.scene.input.MouseEvent;
8  import javafx.scene.layout.Pane;
9  import javafx.scene.paint.Color;
10 import javafx.scene.paint.Paint;
11 import javafx.scene.shape.Circle
12
13 public class PainterController {
```

```
14    // enum representing pen sizes
15    private enum PenSize {
16       SMALL(2),
17       MEDIUM(4),
18       LARGE(6);
19
20       private final int radius;
21
22       PenSize(int radius) {this.radius = radius;} // constructor
23
24       public int getRadius() {return radius;}
25    };
26
```

```
27      // instance variables that refer to GUI components
28      @FXML private RadioButton blackRadioButton;
29      @FXML private RadioButton redRadioButton;
30      @FXML private RadioButton greenRadioButton;
31      @FXML private RadioButton blueRadioButton;
32      @FXML private RadioButton smallRadioButton;
33      @FXML private RadioButton mediumRadioButton;
34      @FXML private RadioButton largeRadioButton;
35      @FXML private Pane drawingAreaPane;
36      @FXML private ToggleGroup colorToggleGroup;
37      @FXML private ToggleGroup sizeToggleGroup;
38
39      // instance variables for managing Painter state
40      private PenSize radius = PenSize.MEDIUM; // radius of circle
41      private Paint brushColor = Color.BLACK; // drawing color
42
```

```
43      // set user data for the RadioButtons
44      @FXML
45      private void initialize() {
46         // user data on a control can be any Object
47         blackRadioButton.setUserData(Color.BLACK);
48         redRadioButton.setUserData(Color.RED);
49         greenRadioButton.setUserData(Color.GREEN);
50         blueRadioButton.setUserData(Color.BLUE);
51         smallRadioButton.setUserData(PenSize.SMALL);
52         mediumRadioButton.setUserData(PenSize.MEDIUM);
53         largeRadioButton.setUserData(PenSize.LARGE);
54      }
55
```

```
56      // handles drawingArea's onMouseDragged MouseEvent
57      @FXML
58      private void drawingAreaMouseDragged(MouseEvent e) {
59         var newCircle = new Circle(e.getX(), e.getY(),
60            radius.getRadius(), brushColor);
61         drawingAreaPane.getChildren().add(newCircle);
62      }
63
```

```
64      // handles color RadioButton's ActionEvents
65      @FXML
66      private void colorRadioButtonSelected(ActionEvent e) {
67         // user data for each color RadioButton is the corresponding Color
68         brushColor =
69            (Color) colorToggleGroup.getSelectedToggle().getUserData();
70      }
71
```

```
72      // handles size RadioButton's ActionEvents
73      @FXML
74      private void sizeRadioButtonSelected(ActionEvent e) {
75         // user data for each size RadioButton is the corresponding PenSize
76         radius =
77            (PenSize) sizeToggleGroup.getSelectedToggle().getUserData();
78      }
79
```

```
80      // handles Undo Button's ActionEvents
81      @FXML
82      private void undoButtonPressed(ActionEvent event) {
83         int count = drawingAreaPane.getChildren().size();
84
85         // if there are any shapes, remove the last one added
86         if (count > 0) {
87            drawingAreaPane.getChildren().remove(count - 1);
88         }
89      }
90
```

```
91      // handles Clear Button's ActionEvents
92      @FXML
93      private void clearButtonPressed(ActionEvent event) {
94         drawingAreaPane.getChildren().clear(); // clear the canvas
95      }
96   }
```

public final void set*PropertyName*(*Type propertyName*)
public final *Type* get*PropertyName*()

https://deitel.com/JavaFXPropertyBindingTutorial

```
1   // Fig. 16.3: ColorChooser.java
2   // Class that loads and displays the ColorChooser's GUI.
3   import javafx.application.Application;
4   import javafx.fxml.FXMLLoader;
5   import javafx.scene.Parent;
6   import javafx.scene.Scene;
7   import javafx.stage.Stage;
8
9   public class ColorChooser extends Application {
10     @Override
11     public void start(Stage stage) throws Exception {
12        Parent root =
13           FXMLLoader.load(getClass().getResource("ColorChooser.fxml"));
14
15        var scene = new Scene(root);
16        stage.setTitle("Color Chooser");
17        stage.setScene(scene);
18        stage.show();
19     }
20
21     public static void main(String[] args) {
22        launch(args);
23     }
24  }
```

```
1   // Fig. 16.4: ColorChooserController.java
2   // Controller for the ColorChooser application
3   import javafx.beans.value.ChangeListener;
4   import javafx.beans.value.ObservableValue;
5   import javafx.fxml.FXML;
6   import javafx.scene.control.Slider;
7   import javafx.scene.control.TextField;
8   import javafx.scene.paint.Color;
9   import javafx.scene.shape.Rectangle;
10
11  public class ColorChooserController {
```

```
12     // instance variables for interacting with GUI components
13     @FXML private Slider redSlider;
14     @FXML private Slider greenSlider;
15     @FXML private Slider blueSlider;
16     @FXML private Slider alphaSlider;
17     @FXML private TextField redTextField;
18     @FXML private TextField greenTextField;
19     @FXML private TextField blueTextField;
20     @FXML private TextField alphaTextField;
21     @FXML private Rectangle colorRectangle;
22
23     // instance variables for managing
24     private int red = 0;
25     private int green = 0;
26     private int blue = 0;
27     private double alpha = 1.0;
28
```

```
29      @FXML
30      private void initialize() {
31         // bind TextField values to corresponding Slider values
32         redTextField.textProperty().bind(
33            redSlider.valueProperty().asString("%.0f"));
34         greenTextField.textProperty().bind(
35            greenSlider.valueProperty().asString("%.0f"));
36         blueTextField.textProperty().bind(
37            blueSlider.valueProperty().asString("%.0f"));
38         alphaTextField.textProperty().bind(
39            alphaSlider.valueProperty().asString("%.2f"));
40
41         // listeners that set Rectangle's fill based on Slider changes
42         redSlider.valueProperty().addListener(
43            (ov, oldValue, newValue) -> {
44               red = newValue.intValue();
45               colorRectangle.setFill(Color.rgb(red, green, blue, alpha));
46            }
47         );
48         greenSlider.valueProperty().addListener(
49            (ov, oldValue, newValue) -> {
50               green = newValue.intValue();
51               colorRectangle.setFill(Color.rgb(red, green, blue, alpha));
52            }
53         );
54         blueSlider.valueProperty().addListener(
55            (ov, oldValue, newValue) -> {
56               blue = newValue.intValue();
57               colorRectangle.setFill(Color.rgb(red, green, blue, alpha));
58            }
59         );
60         alphaSlider.valueProperty().addListener(
61            (ov, oldValue, newValue) -> {
62               alpha = newValue.doubleValue();
63               colorRectangle.setFill(Color.rgb(red, green, blue, alpha));
64            }
65         );
66      }
67   }
```

```
redTextField.textProperty().bind(
   redSlider.valueProperty().asString("%.0f"));
```

```
redSlider.valueProperty().asString("%.0f")
```

```
1   // Fig. 16.5: CoverViewer.java
2   // Class that loads and displays the CoverViewer's GUI.
3   import javafx.application.Application;
4   import javafx.fxml.FXMLLoader;
5   import javafx.scene.Parent;
6   import javafx.scene.Scene;
7   import javafx.stage.Stage;
8
9   public class CoverViewer extends Application {
10     @Override
11     public void start(Stage stage) throws Exception {
12        Parent root =
13           FXMLLoader.load(getClass().getResource("CoverViewer.fxml"));
14
15        var scene = new Scene(root);
16        stage.setTitle("Cover Viewer");
17        stage.setScene(scene);
18        stage.show();
19     }
20
21     public static void main(String[] args) {
22        launch(args);
23     }
24  }
```

```
// Fig. 16.6: CoverViewerController.java
// Controller for Cover Viewer application
import javafx.beans.value.ObservableValue;
import javafx.collections.FXCollections;
import javafx.collections.ObservableList;
import javafx.fxml.FXML;
import javafx.scene.control.ListView;
import javafx.scene.image.Image;
import javafx.scene.image.ImageView;

public class CoverViewerController {
```

```
   // instance variables for interacting with GUI
   @FXML private ListView<Book> booksListView;
   @FXML private ImageView coverImageView;

```

```
   // stores the list of Book Objects
   private final ObservableList<Book> books =
      FXCollections.observableArrayList();

```

```
   // initialize controller
   public void initialize() {
      // populate the ObservableList<Book>
      books.add(new Book("C How to Program",
         "/images/small/chtp.jpg", "/images/large/chtp.jpg"));
      books.add(new Book("C++ How to Program",
         "/images/small/cpphtp.jpg", "/images/large/cpphtp.jpg"));
      books.add(new Book("Intro to Python",
         "/images/small/python.jpg", "/images/large/python.jpg"));
      books.add(new Book("Internet and World Wide Web How to Program",
         "/images/small/iw3htp.jpg", "/images/large/iw3htp.jpg"));
      books.add(new Book("Java How to Program",
         "/images/small/jhtp.jpg", "/images/large/jhtp.jpg"));
      books.add(new Book("Python for Programmers",
         "/images/small/pythonfp.jpg", "/images/large/pythonfp.jpg"));
      books.add(new Book("Visual Basic How to Program",
         "/images/small/vbhtp.jpg", "/images/large/vbhtp.jpg"));
      books.add(new Book("Visual C# How to Program",
         "/images/small/vcshtp.jpg", "/images/large/vcshtp.jpg"));
      booksListView.setItems(books); // bind booksListView to books

      // when ListView selection changes, show large cover in ImageView
      booksListView.getSelectionModel().selectedItemProperty().
         addListener(
            (ov, oldValue, newValue) -> {
               coverImageView.setImage(
                  new Image(newValue.largeImage()));
            }
         );
   }
}
```

```
// Fig. 16.7: ImageTextCell.java
// Custom ListView cell factory that displays an Image and text
import javafx.geometry.Pos;
import javafx.scene.control.Label;
import javafx.scene.control.ListCell;
import javafx.scene.image.Image;
import javafx.scene.image.ImageView;
import javafx.scene.layout.VBox;
import javafx.scene.text.TextAlignment;

public class ImageTextCell extends ListCell<Book> {
   private VBox vbox = new VBox(8.0); // 8 pixels of gap between controls
   private ImageView thumbImageView = new ImageView(); // initially empty
   private Label label = new Label();

```

```
    // constructor configures VBox, ImageView and Label
    public ImageTextCell() {
       vbox.setAlignment(Pos.CENTER); // center VBox contents horizontally

       thumbImageView.setPreserveRatio(true);
       thumbImageView.setFitHeight(100.0); // thumbnail 100 pixels tall
       vbox.getChildren().add(thumbImageView); // attach to Vbox

       label.setWrapText(true); // wrap if text too wide to fit in label
       label.setTextAlignment(TextAlignment.CENTER); // center text
       vbox.getChildren().add(label); // attach to VBox

       setPrefWidth(USE_PREF_SIZE); // use preferred size for cell width
    }

```

```
    // called to configure each custom ListView cell
    @Override
    protected void updateItem(Book item, boolean empty) {
       // required to ensure that cell displays properly
       super.updateItem(item, empty)

       if (empty || item == null) {
          setGraphic(null); // don't display anything
       }
       else {
          // set ImageView's thumbnail image
          thumbImageView.setImage(new Image(item.thumbImage()));
          label.setText(item.title()); // configure Label's text
          setGraphic(vbox); // attach custom layout to ListView cell
       }
    }
}
```

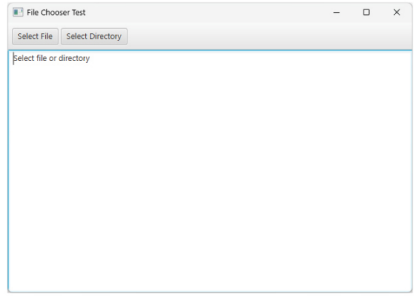
```
// set custom ListView cell factory
booksListView.setCellFactory(
   listView -> new ImageTextCell();
);
```

```
// Fig. 16.8: FileChooserTest.java
// Application to test classes FileChooser and DirectoryChooser.
import javafx.application.Application;
import javafx.fxml.FXMLLoader;
import javafx.scene.Parent;
import javafx.scene.Scene;
import javafx.stage.Stage;

public class FileChooserTest extends Application {
   @Override
   public void start(Stage stage) throws Exception {
      Parent root =
         FXMLLoader.load(getClass().getResource("FileChooserTest.fxml"));

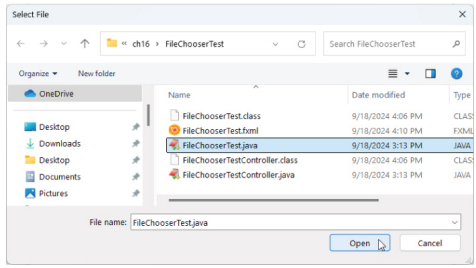
      var scene = new Scene(root);
      stage.setTitle("File Chooser Test"); // displayed in title bar
      stage.setScene(scene);
      stage.show();
   }

   public static void main(String[] args) {
      launch(args);
   }
}
```




```
1  // Fig. 16.9: FileChooserTestController.java
2  // Displays information about a selected file or folder.
3  import java.io.File;
4  import java.io.IOException;
5  import java.nio.file.DirectoryStream;
6  import java.nio.file.Files;
7  import java.nio.file.Path;
8  import java.nio.file.Paths;
9  import javafx.event.ActionEvent;
10 import javafx.fxml.FXML;
11 import javafx.scene.control.Button;
12 import javafx.scene.control.TextArea;
13 import javafx.scene.layout.BorderPane;
14 import javafx.stage.DirectoryChooser;
15 import javafx.stage.FileChooser;
16
17 public class FileChooserTestController {
18    @FXML private BorderPane borderPane;
19    @FXML private Button selectFileButton;
20    @FXML private Button selectDirectoryButton;
21    @FXML private TextArea textArea;
22
```

```
23    // handles selectFileButton's event
24    @FXML
25    private void selectFileButtonPressed(ActionEvent e) {
26       // configure dialog allowing selection of a file
27       var fileChooser = new FileChooser();
28       fileChooser.setTitle("Select File");
29
30       // display files in folder from which the application was launched
31       fileChooser.setInitialDirectory(new File("."));
32
33       // display the FileChooser
34       File file = fileChooser.showOpenDialog(
35          borderPane.getScene().getWindow());
36
37       // process selected Path or display a message
38       if (file != null) {
39          analyzePath(file.toPath());
40       }
41       else {
42          textArea.setText("Select file or directory");
43       }
44    }
45
```



```
46      // handles selectDirectoryButton's event
47      @FXML
48      private void selectDirectoryButtonPressed(ActionEvent e) {
49         // configure dialog allowing selection of a directory
50         var directoryChooser = new DirectoryChooser();
51         directoryChooser.setTitle("Select Directory");
52
53         // display folder from which the application was launched
54         directoryChooser.setInitialDirectory(new File("."));
55
56         // display the DirectoryChooser
57         File file = directoryChooser.showDialog(
58            borderPane.getScene().getWindow());
59
60         // process selected Path or display a message
61         if (file != null) {
62            analyzePath(file.toPath());
63         }
64         else {
65            textArea.setText("Select file or directory");
66         }
67      }
68
```

```
69      // display information about file or directory user specifies
70      public void analyzePath(Path path) {
71         try {
72            // if the file or directory exists, display its info
73            if (path != null && Files.exists(path)) {
74               // gather file (or directory) information
75               var builder = new StringBuilder();
76               builder.append(String.format("%s:%n", path.getFileName()));
77               builder.append(String.format("%s a directory%n",
78                  Files.isDirectory(path) ? "Is" : "Is not"));
79               builder.append(String.format("%s an absolute path%n",
80                  path.isAbsolute() ? "Is" : "Is not"));
81               builder.append(String.format("Last modified: %s%n",
82                  Files.getLastModifiedTime(path)));
83               builder.append(String.format("Size: %s%n", Files.size(path)));
84               builder.append(String.format("Path: %s%n", path));
85               builder.append(String.format("Absolute path: %s%n",
86                  path.toAbsolutePath()));
87
88               if (Files.isDirectory(path)) { // output directory listing
89                  builder.append(String.format("%nDirectory contents:%n"));
90
91                  // object for iterating through a directory's contents
92                  DirectoryStream<Path> directoryStream =
93                     Files.newDirectoryStream(path);
```

```
94
95                  for (Path p : directoryStream) {
96                     builder.append(String.format("%s%n", p));
97                  }
98               }
99
100              // display file or directory info
101              textArea.setText(builder.toString());
102           }
103           else { // Path does not exist
104              textArea.setText("Path does not exist");
105           }
106        }
107        catch (IOException ioException) {
108           textArea.setText(ioException.toString());
109        }
110     }
111  }
```

http://gluonhq.com/products/mobile/

https://github.com/JonathanGiles/scenic-view

https://github.com/mhrimaz/AwesomeJavaFX

https://github.com/webfx-project/webfx

https://controlsfx.github.io/
https://github.com/controlsfx/controlsfx

https://controlsfx.github.io/javadocs/

https://controlsfx.github.io/fxsampler/

https://openjfx.io/javadoc/23/javafx.graphics/javafx/scene/doc-files/cssref.html

```
/* Fig. 17.1: FontsCSS.css */
/* CSS rules that style the VBox and Labels */

.vbox {
   -fx-spacing: 10;
   -fx-padding: 10;
}

```

```
#label1 {
   -fx-font: bold 14pt Arial;
}

```

https://openjfx.io/javadoc/23/javafx.graphics/javafx/scene/doc-files/cssref.html#typefont

https://www.w3.org/TR/selectors-3/

```
#label2 {
   -fx-font: 16pt "Times New Roman";
}

```

-fx-font: 16pt "Times New Roman";

```
#label3 {
   -fx-font: bold italic 16pt "Courier New";
}

```

-fx-font: bold italic 16pt "Courier New";

```
#label5 .text {
   -fx-strikethrough: true;
}

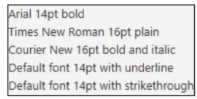
```

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Fig. 17.2: FontCSS.fxml -->
<!-- FontCSS GUI that is styled via external CSS -->

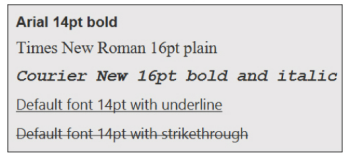
<?import javafx.scene.control.Label?>
<?import javafx.scene.layout.VBox?>

<VBox styleClass="vbox" stylesheets="@FontCSS.css"
   xmlns="http://javafx.com/javafx/23"
   xmlns:fx="http://javafx.com/fxml/1">
   <children>
      <Label fx:id="label1" text="Arial 14pt bold" />
      <Label fx:id="label2" text="Times New Roman 16pt plain" />
      <Label fx:id="label3" text="Courier New 16pt bold and italic" />
      <Label fx:id="label4" text="Default font 14pt with underline" />
      <Label fx:id="label5" text="Default font 14pt with strikethrough" />
   </children>
</VBox>
```

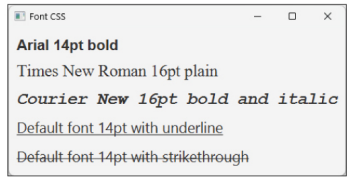
a) GUI as it appears in Scene Builder before referencing the completed CSS file



b) GUI as it appears in Scene Builder after referencing the `FontCSS.css` file containing the rules that style the `VBox` and the `Label`s



c) GUI as it appears in the running program



```
<Label fx:id="label1" text="Arial 14pt bold" />
```

```
<Label fx:id="label1" text="Arial 14pt bold">
</Label>
```

```
xmlns="http://javafx.com/javafx/23"
xmlns:fx="http://javafx.com/fxml/1"
```

```
xmlns="http://javafx.com/javafx/23"
```

```
xmlns:fx="http://javafx.com/fxml/1"
```

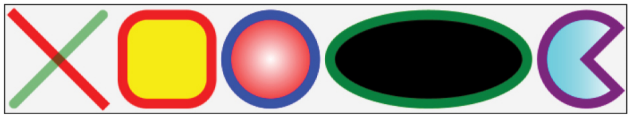
```
scene.getStylesheets().add(
   FontCSS.class.getResource("FontCSS.css").toExternalForm());
```

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Fig. 17.3: BasicShapes.fxml -->
<!-- Defining Shape objects and styling via CSS -->

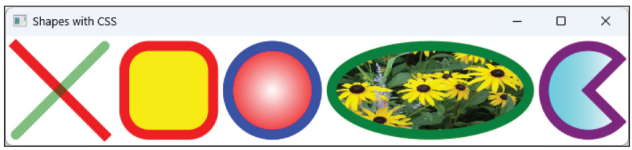
<?import javafx.scene.layout.Pane?>
<?import javafx.scene.shape.Arc?>
<?import javafx.scene.shape.Circle?>
<?import javafx.scene.shape.Ellipse?>
<?import javafx.scene.shape.Line?>
<?import javafx.scene.shape.Rectangle?>

<Pane id="Pane" prefHeight="110.0" prefWidth="630.0"
   stylesheets="@BasicShapes.css" xmlns="http://javafx.com/javafx/23"
   xmlns:fx="http://javafx.com/fxml/1">
   <children>
      <Line fx:id="line1" endX="100.0" endY="100.0"
         startX="10.0" startY="10.0" />
      <Line fx:id="line2" endX="10.0" endY="100.0"
         startX="100.0" startY="10.0" />
      <Rectangle fx:id="rectangle" height="90.0" layoutX="120.0"
         layoutY="10.0" width="90.0" />
      <Circle fx:id="circle" centerX="270.0" centerY="55.0"
         radius="45.0" />
      <Ellipse fx:id="ellipse" centerX="430.0" centerY="55.0"
         radiusX="100.0" radiusY="45.0" />
      <Arc fx:id="arc" centerX="590.0" centerY="55.0" length="270.0"
         radiusX="45.0" radiusY="45.0" startAngle="45.0" type="ROUND" />
   </children>
</Pane>
```

a) GUI in Scene Builder with CSS applied—Ellipse's image fill does not display.



b) GUI in running app—Ellipse's image fill displays correctly.




```
/* Fig. 17.4: BasicShapes.css */
/* CSS that styles various two-dimensional shapes */

Line, Rectangle, Circle, Ellipse, Arc {
   -fx-stroke-width: 10;
}
```

https://openjfx.io/javadoc/23/javafx.graphics/javafx/scene/doc-files/cssref.html#typecolor

```
#line2 {
   -fx-stroke: rgba(0%, 50%, 0%, 0.5);
   -fx-stroke-line-cap: round;
}
```

```
Circle {
   -fx-stroke: blue;
   -fx-fill: radial-gradient(center 50% 50%, radius 60%, white, red);
}
```

```
-fx-fill: radial-gradient(center 50% 50%, radius 60%, white, red);
```

```
29  Ellipse {
30     -fx-stroke: green;
31     -fx-fill: image-pattern("yellowflowers.png");
32  }
33
```

```
34  Arc {
35     -fx-stroke: purple;
36     -fx-fill: linear-gradient(to right, cyan, white);
37  }
```

```
-fx-fill: linear-gradient(to right, cyan, white);
```

https://openjfx.io/javadoc/23/javafx.graphics/javafx/scene/doc-files/cssref.html#typepaint

```
1   // Fig. 17.5: PolyShapesController.java
2   // Drawing Polylines, Polygons and Paths.
3   import javafx.event.ActionEvent;
4   import javafx.fxml.FXML;
5   import javafx.scene.control.RadioButton;
6   import javafx.scene.control.ToggleGroup;
7   import javafx.scene.input.MouseEvent;
8   import javafx.scene.shape.ArcTo;
9   import javafx.scene.shape.ClosePath;
10  import javafx.scene.shape.MoveTo;
11  import javafx.scene.shape.Path;
12  import javafx.scene.shape.Polygon;
13  import javafx.scene.shape.Polyline;
14
15  public class PolyShapesController {
16     // enum representing shape types
17     private enum ShapeType {POLYLINE, POLYGON, PATH};
18
```

```
19     // instance variables that refer to GUI components
20     @FXML private RadioButton polylineRadioButton;
21     @FXML private RadioButton polygonRadioButton;
22     @FXML private RadioButton pathRadioButton;
23     @FXML private ToggleGroup toggleGroup;
24     @FXML private Polyline polyline;
25     @FXML private Polygon polygon;
26     @FXML private Path path;
27
28     // instance variables for managing state
29     private ShapeType shapeType = ShapeType.POLYLINE;
30     private boolean sweepFlag = true; // used with arcs in a Path
31
```

```
32     // set user data for the RadioButtons and display polyline object
33     @FXML
34     private void initialize() {
35        // user data on a control can be any Object
36        polylineRadioButton.setUserData(ShapeType.POLYLINE);
37        polygonRadioButton.setUserData(ShapeType.POLYGON);
38        pathRadioButton.setUserData(ShapeType.PATH);
39
40        displayShape(); // sets polyline's visibility to true when app loads
41     }
42
```

```
43      // handles drawingArea's onMouseClicked event
44      @FXML
45      private void drawingAreaMouseClicked(MouseEvent e) {
46         polyline.getPoints().addAll(e.getX(), e.getY());
47         polygon.getPoints().addAll(e.getX(), e.getY());
48
49         // if path is empty, move to first click position and close path
50         if (path.getElements().isEmpty()) {
51            path.getElements().add(new MoveTo(e.getX(), e.getY()));
52            path.getElements().add(new ClosePath());
53         }
54         else { // insert a new path segment before the ClosePath element
55            // create an arc segment and insert it in the path
56            ArcTo arcTo = new ArcTo();
57            arcTo.setX(e.getX());
58            arcTo.setY(e.getY());
59            arcTo.setRadiusX(100.0);
60            arcTo.setRadiusY(100.0);
61            arcTo.setSweepFlag(sweepFlag);
62            sweepFlag = !sweepFlag;
63            path.getElements().add(path.getElements().size() - 1, arcTo);
64         }
65      }
66
```

```
67      // handles color RadioButton's ActionEvents
68      @FXML
69      private void shapeRadioButtonSelected(ActionEvent e) {
70         // user data for each color RadioButton is a ShapeType constant
71         shapeType =
72            (ShapeType) toggleGroup.getSelectedToggle().getUserData();
73         displayShape(); // display the currently selected shape
74      }
75
```

```
76      // displays currently selected shape
77      private void displayShape() {
78         polyline.setVisible(shapeType == ShapeType.POLYLINE);
79         polygon.setVisible(shapeType == ShapeType.POLYGON);
80         path.setVisible(shapeType == ShapeType.PATH);
81      }
82
```

```
83      // resets each shape
84      @FXML
85      private void clearButtonPressed(ActionEvent event) {
86         polyline.getPoints().clear();
87         polygon.getPoints().clear();
88         path.getElements().clear();
89      }
90   }
```

```java
// Fig. 17.6: DrawStarsController.java
// Create a circle of stars using Polygons and Rotate transforms
import java.util.random.RandomGenerator;
import javafx.fxml.FXML;
import javafx.scene.layout.Pane;
import javafx.scene.paint.Color;
import javafx.scene.shape.Polygon;
import javafx.scene.transform.Transform;

public class DrawStarsController {
   private static final RandomGenerator random =
      RandomGenerator.getDefault();
   @FXML private Pane pane;

   @FXML
   private void initialize() {
      // points that define a five-pointed star shape
      Double[] points = {205.0,150.0, 217.0,186.0, 259.0,186.0,
         223.0,204.0, 233.0,246.0, 205.0,222.0, 177.0,246.0, 187.0,204.0,
         151.0,186.0, 193.0,186.0};

      // create 18 stars
      for (int count = 0; count < 18; ++count) {
         // create a new Polygon and copy existing points into it
         var newStar = new Polygon();

         newStar.getPoints().addAll(points);

         // create random Color and set as newStar's fill
         newStar.setStroke(Color.GREY);
         newStar.setFill(Color.rgb(random.nextInt(256),
            random.nextInt(256), random.nextInt(256),
            random.nextDouble()));

         // apply a rotation to the shape
         newStar.getTransforms().add(
            Transform.rotate(count * 20, 150, 150));
         pane.getChildren().add(newStar);
      }
   }
}
```

https://images.nasa.gov/details/A1Launch

https://convertio.co/video-converter/

https://controlsfx.github.io/
https://github.com/controlsfx/controlsfx

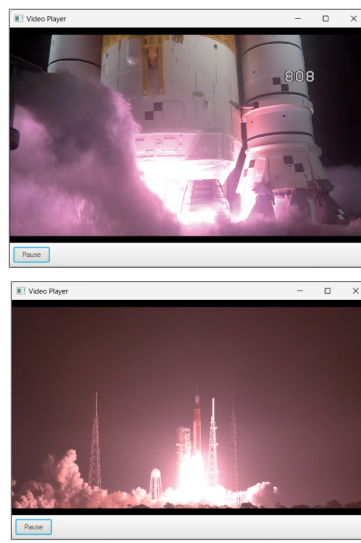
```
javac --module-path pathToFX --add-modules
   javafx.controls,javafx.fxml,javafx.media,org.controlsfx.controls
   *.java
```

```
java --module-path pathToFX --add-modules
   javafx.controls,javafx.fxml,javafx.media,org.controlsfx.controls
   VideoPlayer
```

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Fig. 17.7: VideoPlayer.fxml -->
<!-- VideoPlayer GUI with a MediaView and a Button. -->

<?import javafx.scene.control.Button?>
<?import javafx.scene.control.ToolBar?>
<?import javafx.scene.layout.BorderPane?>
<?import javafx.scene.media.MediaView?>

<BorderPane prefHeight="400.0" prefWidth="600.0"
   style="-fx-background-color: black;"
   xmlns="http://javafx.com/javafx/23"
   xmlns:fx="http://javafx.com/fxml/1"
   fx:controller="VideoPlayerController">
   <bottom>
      <ToolBar prefHeight="40.0" prefWidth="200.0"
         BorderPane.alignment="CENTER">
         <items>
            <Button fx:id="playPauseButton"
              onAction="#playPauseButtonPressed" prefHeight="25.0"
              prefWidth="60.0" text="Play" />
         </items>
      </ToolBar>
   </bottom>
   <center>
      <MediaView fx:id="mediaView" BorderPane.alignment="CENTER" />
   </center>
</BorderPane>
```




```
// Fig. 17.8: VideoPlayerController.java
```

```
1  // Fig. 17.8: VideoPlayerController.java
2  // Using Media, MediaPlayer and MediaView to play a video.
3  import java.net.URL;
4  import javafx.beans.binding.Bindings;
5  import javafx.beans.property.DoubleProperty;
6  import javafx.event.ActionEvent;
7  import javafx.fxml.FXML;
8  import javafx.scene.control.Button;
9  import javafx.scene.media.Media;
10 import javafx.scene.media.MediaPlayer;
11 import javafx.scene.media.MediaPlayer.Status;
12 import javafx.scene.media.MediaView;
13 import javafx.util.Duration;
14 import org.controlsfx.dialog.ExceptionDialog;
15
16 public class VideoPlayerController {
```

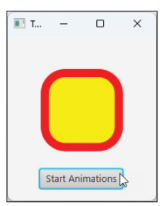
```
17    @FXML private MediaView mediaView;
18    @FXML private Button playPauseButton;
19    private MediaPlayer mediaPlayer;
20
```

```
21    @FXML
22    private void initialize() {
23       // get URL of the video file
24       URL url = VideoPlayerController.class.getResource(
25          "A1Launch~medium.mp4");
26
27       // create a Media object for the specified URL
28       var media = new Media(url.toExternalForm());
29
30       // create a MediaPlayer to control Media playback
31       mediaPlayer = new MediaPlayer(media);
32
33       // specify which MediaPlayer to display in the MediaView
34       mediaView.setMediaPlayer(mediaPlayer);
35
36       // set handler to be called when the video completes playing
37       mediaPlayer.setOnEndOfMedia(() -> {
38          mediaPlayer.seek(Duration.ZERO);
39          mediaPlayer.pause();
40       });
41
42       // set handler that displays an ExceptionDialog if an error occurs
43       mediaPlayer.setOnError(() -> {
44          var dialog = new ExceptionDialog(mediaPlayer.getError());
45          dialog.showAndWait();
46       });
47
48       // set handler to resize mediaView to window size once ready to play
49       mediaPlayer.setOnReady(() -> {
50          DoubleProperty width = mediaView.fitWidthProperty();
51          DoubleProperty height = mediaView.fitHeightProperty();
52          width.bind(Bindings.selectDouble(
53             mediaView.sceneProperty(), "width"));
54          height.bind(Bindings.selectDouble(
55             mediaView.sceneProperty(), "height"));
56       });
57
58       // bind playPauseButton's text to the MediaPlayer's status
59       playPauseButton.textProperty().bind(Bindings.when(
60          mediaPlayer.statusProperty().isEqualTo(Status.PLAYING))
61                      .then("Pause")
62                      .otherwise("Play")
63       );
64    }
65
```

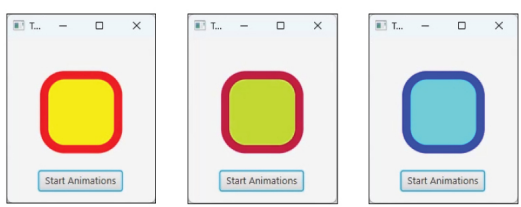
```
66      // toggle media playback and the text on the playPauseButton
67      @FXML
68      private void playPauseButtonPressed(ActionEvent e) {
69         Status status = mediaPlayer.getStatus();
70
71         if (status == Status.READY || status == Status.PAUSED) {
72            mediaPlayer.play();
73         }
74         else {
75            mediaPlayer.pause();
76         }
77      }
78   }
```

```
1   <?xml version="1.0" encoding="UTF-8"?>
2   <!-- Fig. 17.9: TransitionAnimations.fxml -->
3   <!-- FXML for a Rectangle and Button -->
4
5   <?import javafx.scene.control.Button?>
6   <?import javafx.scene.layout.Pane?>
7   <?import javafx.scene.shape.Rectangle?>
8
9   <Pane id="Pane" prefHeight="200.0" prefWidth="180.0"
10     stylesheets="@TransitionAnimations.css"
11     xmlns="http://javafx.com/javafx/23"
12     xmlns:fx="http://javafx.com/fxml/1"
13     fx:controller="TransitionAnimationsController">
14     <children>
15        <Rectangle fx:id="rectangle" height="90.0" layoutX="45.0"
16           layoutY="45.0" width="90.0" />
17        <Button layoutX="38.0" layoutY="161.0" mnemonicParsing="false"
18           onAction="#startButtonPressed" text="Start Animations" />
19     </children>
20  </Pane>
```

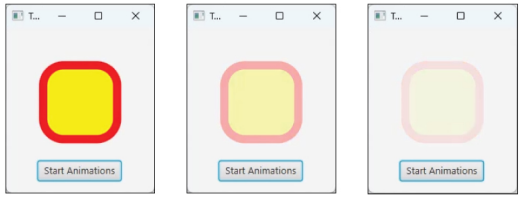
a) Initial **Rectangle**



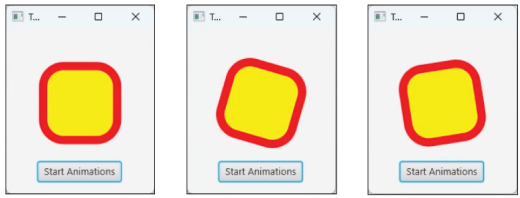
b) **Rectangle** undergoing parallel fill and stroke transitions



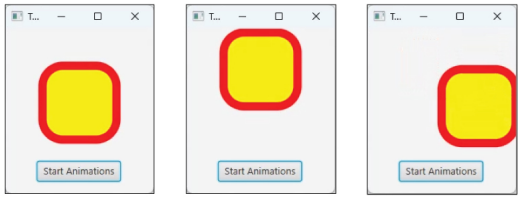
c) **Rectangle** undergoing a fade transition



d) **Rectangle** undergoing a rotate transition



e) **Rectangle** undergoing a path transition



f) **Rectangle** undergoing a scale transition

```
1  // Fig. 17.10: TransitionAnimationsController.java
2  // Applying Transition animations to a Rectangle.
3  import javafx.animation.FadeTransition;
4  import javafx.animation.FillTransition;
5  import javafx.animation.Interpolator;
6  import javafx.animation.ParallelTransition;
7  import javafx.animation.PathTransition;
8  import javafx.animation.RotateTransition;
9  import javafx.animation.ScaleTransition;
10 import javafx.animation.SequentialTransition;
11 import javafx.animation.StrokeTransition;
12 import javafx.event.ActionEvent;
13 import javafx.fxml.FXML;
14 import javafx.scene.paint.Color;
15 import javafx.scene.shape.LineTo;
16 import javafx.scene.shape.MoveTo;
17 import javafx.scene.shape.Path;
18 import javafx.scene.shape.Rectangle;
19 import javafx.util.Duration;
20
21 public class TransitionAnimationsController {
22    @FXML private Rectangle rectangle;
23
```

```
24      // configure and start transition animations
25      @FXML
26      private void startButtonPressed(ActionEvent event) {
27         // transition that changes a shape's fill
28         var fillTransition = new FillTransition(Duration.seconds(1));
29         fillTransition.setToValue(Color.CYAN);
30         fillTransition.setCycleCount(2);
31
32         // each even cycle plays transition in reverse to restore original
33         fillTransition.setAutoReverse(true);
34
```

```
35         // transition that changes a shape's stroke over time
36         var strokeTransition = new StrokeTransition(Duration.seconds(1));
37         strokeTransition.setToValue(Color.BLUE);
38         strokeTransition.setCycleCount(2);
39         strokeTransition.setAutoReverse(true);
40
```

```
41         // parallelizes multiple transitions
42         var parallelTransition =
43            new ParallelTransition(fillTransition, strokeTransition);
44
```

```
45         // transition that changes a node's opacity over time
46         var fadeTransition = new FadeTransition(Duration.seconds(1));
47         fadeTransition.setFromValue(1.0); // opaque
48         fadeTransition.setToValue(0.0); // transparent
49         fadeTransition.setCycleCount(2);
50         fadeTransition.setAutoReverse(true);
51
```

```
52         // transition that rotates a node
53         var rotateTransition = new RotateTransition(Duration.seconds(1));
54         rotateTransition.setByAngle(360.0);
55         rotateTransition.setCycleCount(2);
56         rotateTransition.setInterpolator(Interpolator.EASE_BOTH);
57         rotateTransition.setAutoReverse(true);
58
```

https://openjfx.io/javadoc/23/javafx.graphics/javafx/animation/Interpolator.html

```
59         // transition that moves a node along a Path
60         var path = new Path(new MoveTo(45, 45), new LineTo(45, 0),
61            new LineTo(90, 0), new LineTo(90, 90), new LineTo(0, 90));
62         var translateTransition =
63            new PathTransition(Duration.seconds(2), path);
64         translateTransition.setCycleCount(2);
65         translateTransition.setInterpolator(Interpolator.EASE_IN);
66         translateTransition.setAutoReverse(true);
67
```

```
68         // transition that scales a shape to make it larger or smaller
69         var scaleTransition = new ScaleTransition(Duration.seconds(1));
70         scaleTransition.setByX(0.75);
71         scaleTransition.setByY(0.75);
72         scaleTransition.setCycleCount(2);
73         scaleTransition.setInterpolator(Interpolator.EASE_OUT);
74         scaleTransition.setAutoReverse(true);
75
```

```
        // transition that applies a sequence of transitions to a node
        var sequentialTransition = new SequentialTransition(
           rectangle, parallelTransition, fadeTransition,
           rotateTransition, translateTransition, scaleTransition);
        sequentialTransition.play(); // play the transition
     }
  }
```

```
  <?xml version="1.0" encoding="UTF-8"?>
  <!-- Fig. 17.11: TimelineAnimation.fxml -->
  <!-- FXML for a Circle that will be animated by the controller -->

  <?import javafx.scene.layout.Pane?>
  <?import javafx.scene.shape.Circle?>

  <Pane id="Pane" fx:id="pane" prefHeight="400.0"
     prefWidth="600.0" xmlns:fx="http://javafx.com/fxml/1"
     xmlns="http://javafx.com/javafx/23"
     fx:controller="TimelineAnimationController">
     <children>
        <Circle fx:id="c" fill="DODGERBLUE" layoutX="142.0" layoutY="143.0"
           radius="40.0" stroke="BLACK" strokeType="INSIDE"
           strokeWidth="5.0" />
     </children>
  </Pane>
```

```
  // Fig. 17.12: TimelineAnimationController.java
  // Bounce a circle around a window using a Timeline animation
  import java.util.random.RandomGenerator;
  import javafx.animation.KeyFrame;
  import javafx.animation.Timeline;
  import javafx.fxml.FXML;
  import javafx.geometry.Bounds;
  import javafx.scene.layout.Pane;
  import javafx.scene.shape.Circle;
  import javafx.util.Duration;

  public class TimelineAnimationController {
     private final static RandomGenerator random =
        RandomGenerator.getDefault();
     private int dx = random.nextInt(1, 6);
     private int dy = random.nextInt(1, 6);
     @FXML private Circle c;
     @FXML private Pane pane;

     @FXML
     private void initialize() {
        // define a timeline animation
        var timelineAnimation = new Timeline(
           new KeyFrame(Duration.millis(10), e -> {
              // move the circle by the dx and dy amounts
              c.setLayoutX(c.getLayoutX() + dx);
              c.setLayoutY(c.getLayoutY() + dy);
              Bounds bounds = pane.getBoundsInLocal();

              if (hitRightOrLeftEdge(bounds)) {
                 dx *= -1;
              }

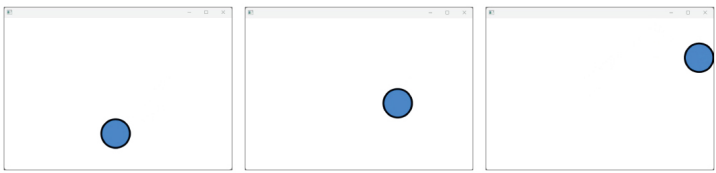
              if (hitTopOrBottom(bounds)) {
```

```
            dy *= -1;
         }
      }) // end call to KeyFrame constructor
   );

   // indicate that the timeline animation should run indefinitely
   timelineAnimation.setCycleCount(Timeline.INDEFINITE);
   timelineAnimation.play();
}

// determines whether the circle hit the left or right of the window
private boolean hitRightOrLeftEdge(Bounds bounds) {
   return (c.getLayoutX() <= (bounds.getMinX() + c.getRadius())) ||
      (c.getLayoutX() >= (bounds.getMaxX() - c.getRadius()));
}

// determines whether the circle hit the top or bottom of the window
private boolean hitTopOrBottom(Bounds bounds) {
   return (c.getLayoutY() <= (bounds.getMinY() + c.getRadius())) ||
      (c.getLayoutY() >= (bounds.getMaxY() - c.getRadius()));
}
}
```



```
// Fig. 17.13: BallAnimationTimerController.java
// Bounce a circle around a window using an AnimationTimer subclass.
import java.util.random.RandomGenerator;
import javafx.animation.AnimationTimer;
import javafx.fxml.FXML;
import javafx.geometry.Bounds;
import javafx.scene.layout.Pane;
import javafx.scene.shape.Circle;

public class BallAnimationTimerController {
   private final static RandomGenerator random =
      RandomGenerator.getDefault();
   private int dx = random.nextInt(1, 6);
   private int dy = random.nextInt(1, 6);
   @FXML private Circle c;
   @FXML private Pane pane;

   @FXML
   private void initialize() {
      // define an AnimationTimer
      AnimationTimer timer = new AnimationTimer() {
         int velocity = 60; // used to scale distance changes
         long previousTime = System.nanoTime(); // time since app launch

         // specify how to move Circle for current animation frame
         @Override
         public void handle(long now) {
            double elapsedTime = (now - previousTime) / 1000000000.0;
            previousTime = now;
            double scale = elapsedTime * velocity;

            Bounds bounds = pane.getBoundsInLocal();
            c.setLayoutX(c.getLayoutX() + dx * scale);
            c.setLayoutY(c.getLayoutY() + dy * scale);

```

```
            if (hitRightOrLeftEdge(bounds)) {
               dx *= -1;
            }

            if (hitTopOrBottom(bounds)) {
               dy *= -1;
            }
         }
      };

      timer.start();
   }

   // determines whether the circle hit left/right of the window
   private boolean hitRightOrLeftEdge(Bounds bounds) {
      return (c.getLayoutX() <= (bounds.getMinX() + c.getRadius())) ||
         (c.getLayoutX() >= (bounds.getMaxX() - c.getRadius()));
   }

   // determines whether the circle hit top/bottom of the window
   private boolean hitTopOrBottom(Bounds bounds) {
      return (c.getLayoutY() <= (bounds.getMinY() + c.getRadius())) ||
         (c.getLayoutY() >= (bounds.getMaxY() - c.getRadius()));
   }
}
```

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Fig. 17.14: CSSTransition.fxml -->
<!-- FXML for a Pane and a Rectangle -->

<?import javafx.scene.layout.Pane?>
<?import javafx.scene.shape.Rectangle?>

<Pane prefHeight="150.0" prefWidth="150.0"
   stylesheets="@CSSTransition.css"
   xmlns="http://javafx.com/javafx/23"
   xmlns:fx="http://javafx.com/fxml/1">
   <children>
      <Rectangle fx:id="rectangle" height="90.0"
         layoutX="30.0" layoutY="30.0" width="90.0" />
   </children>
</Pane>
```

```
/* Fig. 17.15: CSSTransition.css */
/* CSS rules to apply styling and CSS transitions to a Rectangle. */
Rectangle {
   -fx-stroke-width: 5;
   -fx-stroke: black;
   -fx-arc-width: 50;
   -fx-arc-height: 50;
   -fx-fill: red;
   -fx-rotate: 0;
   transition-property: -fx-fill, -fx-stroke,
      -fx-stroke-width, -fx-rotate;
   transition-duration: 3.0s;
}

Rectangle:hover {
   -fx-stroke-width: 15;
   -fx-stroke: red;
   -fx-fill: black;
   -fx-rotate: 180;
}
```

transition-timing-function: *value*;

https://openjfx.io/javadoc/23/javafx.graphics/javafx/scene/doc-files/cssref.html#typeeasingfunction

```
transition:
    property duration timing-function delay,
    ...
    property duration timing-function delay;
```

```
transition:
    -fx-fill 3.0s ease 0s,
    -fx-stroke 3.0s ease 0s,
    -fx-stroke-width 3.0s ease 0s,
    -fx-rotate 3.0s ease 0s;
```

https://dev.to/smpnjn/css-transition-generator-5de0

```
// Fig. 17.16: CanvasShapesController.java
// Drawing on a Canvas.
import javafx.fxml.FXML;
import javafx.scene.canvas.Canvas;
import javafx.scene.canvas.GraphicsContext;
import javafx.scene.image.Image;
import javafx.scene.paint.Color;
import javafx.scene.paint.CycleMethod;
import javafx.scene.paint.ImagePattern;
import javafx.scene.paint.LinearGradient;
import javafx.scene.paint.RadialGradient;
import javafx.scene.paint.Stop;
import javafx.scene.shape.ArcType;
import javafx.scene.shape.StrokeLineCap;

public class CanvasShapesController {
   // instance variables that refer to GUI components
   @FXML private Canvas drawingCanvas;

```

```
   // draw on the Canvas
   @FXML
   private void initialize() {
      GraphicsContext gc = drawingCanvas.getGraphicsContext2D();
      gc.setLineWidth(10); // set all stroke widths

```

```
      // draw red line
      gc.setStroke(Color.RED);
      gc.strokeLine(10, 10, 100, 100);

      // draw green line
      gc.setGlobalAlpha(0.5); // half transparent
      gc.setLineCap(StrokeLineCap.ROUND);
      gc.setStroke(Color.GREEN);
      gc.strokeLine(100, 10, 10, 100);

      gc.setGlobalAlpha(1.0); // reset alpha transparency

```

```
      // draw rounded rect with red border and yellow fill
      gc.setStroke(Color.RED);
      gc.setFill(Color.YELLOW);
      gc.fillRoundRect(120, 10, 90, 90, 50, 50);
      gc.strokeRoundRect(120, 10, 90, 90, 50, 50);

```

```
        // draw circle with blue border and red/white radial gradient fill
        gc.setStroke(Color.BLUE);
        Stop[] stopsRadial =
           {new Stop(0, Color.RED), new Stop(1, Color.WHITE)};
        var radialGradient = new RadialGradient(0, 0, 0.5, 0.5,
           0.6, true, CycleMethod.NO_CYCLE, stopsRadial);
        gc.setFill(radialGradient);
        gc.fillOval(230, 10, 90, 90);
        gc.strokeOval(230, 10, 90, 90);

        // draw ellipse with green border and image fill
        gc.setStroke(Color.GREEN);
        gc.setFill(new ImagePattern(new Image("yellowflowers.png")));
        gc.fillOval(340, 10, 200, 90);
        gc.strokeOval(340, 10, 200, 90);

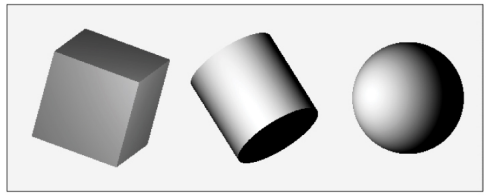
        // draw arc with purple border and cyan/white linear gradient fill
        gc.setStroke(Color.PURPLE);
        Stop[] stopsLinear =
           {new Stop(0, Color.CYAN), new Stop(1, Color.WHITE)};
        var linearGradient = new LinearGradient(0, 0, 1, 0,
           true, CycleMethod.NO_CYCLE, stopsLinear);
        gc.setFill(linearGradient);
        gc.fillArc(560, 10, 90, 90, 45, 270, ArcType.ROUND);
        gc.strokeArc(560, 10, 90, 90, 45, 270, ArcType.ROUND);
    }
}
```

https://openjfx.io/javadoc/23/javafx.graphics/javafx/scene/canvas/GraphicsContext.html

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Fig. 17.17: ThreeDimensionalShapes.fxml -->
<!-- FXML that displays a Box, Cylinder and Sphere -->

<?import javafx.geometry.Point3D?>
<?import javafx.scene.layout.Pane?>
<?import javafx.scene.shape.Box?>
<?import javafx.scene.shape.Cylinder?>
<?import javafx.scene.shape.Sphere?>

<Pane prefHeight="200.0" prefWidth="510.0"
   xmlns="http://javafx.com/javafx/23"
   xmlns:fx="http://javafx.com/fxml/1"
   fx:controller="ThreeDimensionalShapesController">
   <children>
      <Box fx:id="box" depth="100.0" height="100.0" layoutX="100.0"
         layoutY="100.0" rotate="30.0" width="100.0">
         <rotationAxis>
            <Point3D x="1.0" y="1.0" z="1.0" />
         </rotationAxis>
      </Box>
      <Cylinder fx:id="cylinder" height="100.0" layoutX="265.0"
         layoutY="100.0" radius="50.0" rotate="-45.0">
         <rotationAxis>
            <Point3D x="1.0" y="1.0" z="1.0" />
         </rotationAxis>
      </Cylinder>
      <Sphere fx:id="sphere" layoutX="430.0" layoutY="100.0"
         radius="60.0" />
   </children>
</Pane>
```



https://en.wikipedia.org/wiki/Phong_shading

```java
// Fig. 17.18: ThreeDimensionalShapesController.java
// Setting the material displayed on 3D shapes.
import javafx.fxml.FXML;
import javafx.scene.paint.Color;
import javafx.scene.paint.PhongMaterial;
import javafx.scene.image.Image;
import javafx.scene.shape.Box;
import javafx.scene.shape.Cylinder;
import javafx.scene.shape.Sphere;

public class ThreeDimensionalShapesController {
   // instance variables that refer to 3D shapes
   @FXML private Box box;
   @FXML private Cylinder cylinder;
   @FXML private Sphere sphere;

   // set the material for each 3D shape
   @FXML
   private void initialize() {
      // define material for the Box object
      var boxMaterial = new PhongMaterial();
      boxMaterial.setDiffuseColor(Color.CYAN);
      box.setMaterial(boxMaterial);

      // define material for the Cylinder object
      var cylinderMaterial = new PhongMaterial();
      cylinderMaterial.setDiffuseMap(new Image("yellowflowers.png"));
      cylinder.setMaterial(cylinderMaterial);

      // define material for the Sphere object
      var sphereMaterial = new PhongMaterial();
      sphereMaterial.setDiffuseColor(Color.RED);
      sphereMaterial.setSpecularColor(Color.WHITE);
      sphereMaterial.setSpecularPower(32);
      sphere.setMaterial(sphereMaterial);

   }
}
```

https://github.com/AlmasB/FXGL

https://github.com/AlmasB/FXGL/tree/dev/fxgl-samples/src/main/java

https://github.com/AlmasB/FXGLGames

https://www.youtube.com/
playlist?list=PL4h6ypqTi3RTiTuAQFKE6xwflnPKyFuPp

https://github.com/AlmasB/FXGL/wiki/FXGL-11

https://www.jfx-central.com/
https://foojay.io/today/category/javafx/

```
1  // Fig. 18.1: SortComparison.java
2  // Comparing the performance of Arrays methods sort and parallelSort.
3  import java.time.Duration;
4  import java.time.InstantSource;
5  import java.text.NumberFormat;
6  import java.util.Arrays;
7  import java.util.concurrent.ThreadLocalRandom;
8
9  public class SortComparison {
10    public static void main(String[] args) {
11       var clock = InstantSource.system(); // system clock for timing sorts
12
```

ThreadLocalRandom.current().nextInt()

```
13       // create array of random ints, then copy it; Arrays method
14       // parallelSetAll divides the task of randomly initializing
15       // array1 across available cores
16       var array1 = new int[100_000_000];
17       Arrays.parallelSetAll(array1,
18          i -> ThreadLocalRandom.current().nextInt());
19       var array2 = array1.clone();
20
```

```
21       // time the sorting of array1 with Arrays method sort
22       System.out.println("Starting sort");
23       var sortStart = clock.instant();
24       Arrays.sort(array1);
25       var sortEnd = clock.instant();
26
27       // display timing results
28       long sortTime = Duration.between(sortStart, sortEnd).toMillis();
29       System.out.printf("Sort time: %.3f seconds%n%n", sortTime / 1000.0);
30
```

```
Starting sort
Sort time: 9.100 seconds
```

```
31       // time the sorting of array2 with Arrays method parallelSort
32       System.out.println("Starting parallelSort");
33       var parallelSortStart = clock.instant();
34       Arrays.parallelSort(array2);
35       var parallelSortEnd = clock.instant();
36
37       // display timing results
38       long parallelSortTime =
39          Duration.between(parallelSortStart, parallelSortEnd).toMillis();
40       System.out.printf("Sort time: %.3f seconds%n%n",
41          parallelSortTime / 1000.0);
42
```

```
Starting parallelSort
Sort time: 0.965 seconds
```

```
43       // display time difference as a percentage
44       String percentage = NumberFormat.getPercentInstance().format(
45          (double) (sortTime - parallelSortTime) / parallelSortTime);
46       System.out.printf("sort took %s more time than parallelSort%n",
47          percentage);
48    }
49 }
```

```
sort took 843% more time than parallelSort
```

```
1  // Fig. 18.2: StreamStatisticsComparison.java
2  // Comparing performance of sequential and parallel stream operations.
3  import java.time.Duration;
4  import java.time.InstantSource;
5  import java.util.Arrays;
6  import java.util.LongSummaryStatistics;
7  import java.util.concurrent.ThreadLocalRandom;
8
9  public class StreamStatisticsComparison {
10    public static void main(String[] args) {
11       var clock = InstantSource.system(); // system clock for timing
12
13       // create array of random long values
14       var values = new long[1_000_000_000];
15       Arrays.parallelSetAll(values,
16          i -> ThreadLocalRandom.current().nextLong(1, 1001));
17
```

```
18       // perform calculations separately
19       var separateStart = clock.instant();
20       long count = Arrays.stream(values).count();
21       long sum = Arrays.stream(values).sum();
22       long min = Arrays.stream(values).min().getAsLong();
23       long max = Arrays.stream(values).max().getAsLong();
24       double average = Arrays.stream(values).average().getAsDouble();
25       var separateEnd = clock.instant();
26
27       // display results
28       System.out.println("Calculations performed separately");
29       System.out.printf("    count: %,d%n", count);
30       System.out.printf("      sum: %,d%n", sum);
31       System.out.printf("      min: %,d%n", min);
32       System.out.printf("      max: %,d%n", max);
33       System.out.printf("  average: %f%n", average);
34       System.out.printf("Total time in milliseconds: %d%n%n",
35          Duration.between(separateStart, separateEnd).toMillis());
36
```

```
Calculations performed separately
    count: 1,000,000,000
      sum: 500,490,878,099
      min: 1
      max: 1,000
  average: 500.490878
Total time in milliseconds: 4052
```

```
37       // time summaryStatistics operation with sequential stream
38       System.out.println("Calculating statistics on sequential stream");
39       var sequentialStart = clock.instant();
40       LongSummaryStatistics results1 =
41          Arrays.stream(values).summaryStatistics();
42       var sequentialEnd = clock.instant();
43
44       // display results
45       displayStatistics(results1);
46       System.out.printf("Total time in milliseconds: %d%n%n",
47          Duration.between(sequentialStart, sequentialEnd).toMillis());
48
```

```
Calculating statistics on sequential stream
Statistics
    count: 1,000,000,000
      sum: 500,490,878,099
      min: 1
      max: 1,000
  average: 500.490878
Total time in milliseconds: 3451
```

```
        // time sum operation with parallel stream
        System.out.println("Calculating statistics on parallel stream");
        var parallelStart = clock.instant();
        LongSummaryStatistics results2 =
           Arrays.stream(values).parallel().summaryStatistics();
        var parallelEnd = clock.instant();

        // display results
        displayStatistics(results2);
        System.out.printf("Total time in milliseconds: %d%n%n",
           Duration.between(parallelStart, parallelEnd).toMillis());
     }

```

```
Calculating statistics on parallel stream
Statistics
    count: 1,000,000,000
      sum: 500,490,878,099
      min: 1
      max: 1,000
  average: 500.490878
Total time in milliseconds: 359
```

```
     // display's LongSummaryStatistics values
     private static void displayStatistics(LongSummaryStatistics stats) {
        System.out.println("Statistics");
        System.out.printf("    count: %,d%n", stats.getCount());
        System.out.printf("      sum: %,d%n", stats.getSum());
        System.out.printf("      min: %,d%n", stats.getMin());
        System.out.printf("      max: %,d%n", stats.getMax());
        System.out.printf("  average: %f%n", stats.getAverage());
     }
  }
```

```
  // Fig. 18.3: PrintTask.java
  // PrintTask class sleeps for a random time from 0 to 5 seconds
  import java.util.random.RandomGenerator;

  public class PrintTask implements Runnable {
     private static final RandomGenerator generator =
        RandomGenerator.getDefault();
     private final int sleepTime; // random sleep time for thread
     private final String taskName;

     // constructor gives the PrintTask a name and
     // picks random sleep time between 0 and 5 seconds
     public PrintTask(String taskName) {
        this.taskName = taskName;
        sleepTime = generator.nextInt(5001); // milliseconds
     }

     // method run contains the code that a thread will execute
     @Override
     public void run() {
        try { // put thread to sleep for sleepTime amount of time
           System.out.printf("%s going to sleep for %d milliseconds.%n",
              taskName, sleepTime);
           Thread.sleep(sleepTime); // put thread to sleep
        }
        catch (InterruptedException exception) {
           exception.printStackTrace();
           Thread.currentThread().interrupt(); // re-interrupt the thread
        }

        // print task name
        System.out.printf("%s done sleeping%n", taskName);
     }
  }
```

```java
// Fig. 18.4: TaskExecutor.java
// Using an ExecutorService to execute Runnables.
import java.util.concurrent.Executors;

public class TaskExecutor {
   public static void main(String[] args) {
      // create and name each runnable
      var task1 = new PrintTask("task1");
      var task2 = new PrintTask("task2");
      var task3 = new PrintTask("task3");

      System.out.println("Starting Executor\n");

      // create ExecutorService for executing tasks; when the
      // try-with-resources statement completes, the ExecutorService's
      // close method shuts it down, preventing it from accepting new
      // tasks and waiting for existing ones to complete execution
      try (var executorService = Executors.newCachedThreadPool()) {
         // start the three PrintTasks
         executorService.execute(task1); // start task1
         executorService.execute(task2); // start task2
         executorService.execute(task3); // start task3
      }


      System.out.println("\nMain ends.");
   }
}
```

```
Starting Executor

task1 going to sleep for 4610 milliseconds.
task3 going to sleep for 4833 milliseconds.
task2 going to sleep for 992 milliseconds.
task2 done sleeping
task1 done sleeping
task3 done sleeping

Main ends.
```

```
Starting Executor

task3 going to sleep for 357 milliseconds.
task1 going to sleep for 4670 milliseconds.
task2 going to sleep for 3360 milliseconds.
task3 done sleeping
task2 done sleeping
task1 done sleeping

Main ends.
```

```java
// Fig. 18.5: VirtualThreadDemo.java
// Using virtual threads to execute Runnables.
import java.util.concurrent.Executors;

public class VirtualThreadDemo {
   public static void main(String[] args) {
      // PrintTask constructor assigns random sleep time to each
      PrintTask task1 = new PrintTask("task1");
      PrintTask task2 = new PrintTask("task2");
      PrintTask task3 = new PrintTask("task3");

      System.out.println("Starting Virtual-Thread-Per-Task Executor\n");

      // The ExecutorService returned by newVirtualThreadPerTaskExecutor
      // runs each task in a virtual thread. The try-with-resources block
      // calls ExecutorService's close method when the block terminates.
      // The program then waits for the virtual threads to complete.
      try (var executor = Executors.newVirtualThreadPerTaskExecutor()) {
         // start the three PrintTasks
         executor.execute(task1); // start task1
         executor.execute(task2); // start task2
         executor.execute(task3); // start task3
      }

```

```
25          System.out.println("\nMain ends.");
26      }
27  }
```

```
Starting Virtual-Thread-Per-Task Executor

task2 going to sleep for 2138 milliseconds.
task3 going to sleep for 4525 milliseconds.
task1 going to sleep for 90 milliseconds.
task1 done sleeping
task2 done sleeping
task3 done sleeping

Main ends.
```

```
Starting Virtual-Thread-Per-Task Executor

task1 going to sleep for 3128 milliseconds.
task2 going to sleep for 3661 milliseconds.
task3 going to sleep for 1080 milliseconds.
task3 done sleeping
task1 done sleeping
task2 done sleeping

Main ends.
```

```
var virtualThread = Thread.startVirtualThread(task1);
```

```
1   // Fig. 18.6: PlatformAndVirtualThreads.java
2   // Timing large numbers of platform vs. virtual threads.
3   import java.time.Duration;
4   import java.time.InstantSource;
5   import java.util.ArrayList;
6   import java.util.concurrent.ExecutorService;
7   import java.util.concurrent.Executors;
8   import java.util.concurrent.Future;
9   import java.util.stream.LongStream;
10
11  public class PlatformAndVirtualThreads {
```

```
12      // execute and time threads; sum squares of 1 to number of tasks
13      private static void executeAndTimeThreads(int tasks,
14         String threadType, ExecutorService executor) throws Exception {
15         var clock = InstantSource.system(); // system clock for timing
16
17         // stores Futures for all the launched tasks
18         var futures = new ArrayList<Future<Long>>(tasks);
19
20         System.out.printf(
21            "Executing %d tasks on %s...%n", tasks, threadType);
22         var start = clock.instant();
23
24         // auto-closes the ExecutorService at the end of the try block;
25         // automatically waits for all tasks to complete
26         try (executor) {
27            // launch the number of tasks specified by tasks parameter
28            LongStream.rangeClosed(1, tasks).forEach(value ->
29               futures.add(executor.submit(() -> {
30                  Thread.sleep(Duration.ofSeconds(1)); // simulate blocking
31                  return value * value;
32               }))
33            );
34         }
35
36         // sum the results from the Futures
37         long sum = 0;
38         for (var future : futures) {
39            sum += future.get();
40         }
41
42         var end = clock.instant();
43
44         // display results
45         System.out.printf("Execution time: %.2f seconds%n",
46            Duration.between(start, end).toMillis() / 1000.0);
47         System.out.printf("Sum of squares 1-%d: %d%n%n", tasks, sum);
48      }
49
```

```
50      public static void main(String[] args) throws Exception {
51         // check for proper command-line arguments
52         if (args.length != 1) {
53            System.out.println("Usage: PlatformAndVirtualThreads <count>");
54            System.exit(1);
55         }
56
57         // get number of threads to launch
58         final int TASKS = Integer.parseInt(args[0]);
59
60         // execute and time platform threads with fixed thread pool
61         executeAndTimeThreads(TASKS,
62            "Platform Threads (fixed pool of 1000 threads)",
63            Executors.newFixedThreadPool(1000));
64
65         // execute and time platform threads with cached thread pool
66         executeAndTimeThreads(TASKS,
67            "Platform Threads (cached thread pool)",
68            Executors.newCachedThreadPool());
69
70         // execute and time virtual threads
71         executeAndTimeThreads(TASKS,
72            "Virtual Threads", Executors.newVirtualThreadPerTaskExecutor());
73      }
74   }
```

```
java PlatformAndVirtualThreads numberOfTasks
```

```
Executing 1000 tasks on Platform Threads (fixed pool of 1000 threads)...
Execution time: 1.04 seconds
Sum of squares 1-1000: 333833500

Executing 1000 tasks on Platform Threads (cached thread pool)...
Execution time: 1.03 seconds
Sum of squares 1-1000: 333833500

Executing 1000 tasks on Virtual Threads...
Execution time: 1.02 seconds
Sum of squares 1-1000: 333833500
```

```
Executing 5000 tasks on Platform Threads (fixed pool of 1000 threads)...
Execution time: 5.07 seconds
Sum of squares 1-5000: 41679167500

Executing 5000 tasks on Platform Threads (cached thread pool)...
Execution time: 1.33 seconds
Sum of squares 1-5000: 41679167500

Executing 5000 tasks on Virtual Threads...
Execution time: 1.05 seconds
Sum of squares 1-5000: 41679167500
```

```
Executing 10000 tasks on Platform Threads (fixed pool of 1000 threads)...
Execution time: 10.08 seconds
Sum of squares 1-10000: 333383335000

Executing 10000 tasks on Platform Threads (cached thread pool)...
Execution time: 1.93 seconds
Sum of squares 1-10000: 333383335000

Executing 10000 tasks on Virtual Threads...
Execution time: 1.07 seconds
Sum of squares 1-10000: 333383335000
```

```
Executing 50000 tasks on Platform Threads (fixed pool of 1000 threads)...
Execution time: 50.26 seconds
Sum of squares 1-50000: 41667916675000

Executing 50000 tasks on Platform Threads (cached thread pool)...
Execution time: 9.08 seconds
Sum of squares 1-50000: 41667916675000

Executing 50000 tasks on Virtual Threads...
Execution time: 1.54 seconds
Sum of squares 1-50000: 41667916675000
```

```
javac --enable-preview -source 23 -cp ".:../libs/*" WeatherApp.java
```

```
java --enable-preview -cp ".:../libs/*" WeatherApp Boston \
   "New Orleans" "Los Angeles"
```

```
1  // Fig. 18.7: WeatherApp.java
2  // Fetch a city's weather using OpenWeatherMap API with
3  // structured concurrency and scoped values.
4  import com.fasterxml.jackson.annotation.JsonIgnoreProperties;
5  import com.fasterxml.jackson.databind.ObjectMapper;
6  import java.lang.ScopedValue;
7  import java.net.URI;
8  import java.net.URLEncoder;
9  import java.net.http.HttpClient;
10 import java.net.http.HttpRequest;
11 import java.net.http.HttpResponse;
12 import java.nio.charset.StandardCharsets;
13 import java.time.Instant;
14 import java.time.ZoneId;
15 import java.time.ZoneOffset;
16 import java.time.format.DateTimeFormatter;
17 import java.util.List;
18 import java.util.concurrent.Executors;
19 import java.util.concurrent.StructuredTaskScope;
20 import java.util.stream.Collectors;
21
```

https://openweathermap.org/current

https://openweathermap.org/forecast5

```
22 // record classes for current weather JSON mapping
23 @JsonIgnoreProperties(ignoreUnknown = true)
24 record WeatherData(Weather[] weather, Main main, long dt,
25    int timezone, String name) {}
26
27 @JsonIgnoreProperties(ignoreUnknown = true)
28 record Weather(String main, String description) {}
29
30 @JsonIgnoreProperties(ignoreUnknown = true)
31 record Main(double temp) {}
32
33 // record classes for forecast JSON mapping
34 @JsonIgnoreProperties(ignoreUnknown = true)
35 record ForecastResponse(List<ForecastEntry> list) {}
36
37 @JsonIgnoreProperties(ignoreUnknown = true)
38 record ForecastEntry(long dt, Main temp, Weather[] weather) {}
39
```

```
40 public class WeatherApp {
41    // load OpenWeatherMap API key
42    private static final String API_KEY =
43       System.getenv("OPENWEATHERMAP_API_KEY");
44
45    // HttpClient for making web-service requests
46    private static final HttpClient HTTP_CLIENT =
47       HttpClient.newHttpClient();
48
49    // used to launch each group of requests for a specified city
50    private static final ScopedValue<String> CITY =
51       ScopedValue.newInstance();
52
```

```
53      public static void main(String[] args) throws Exception {
54         // check for proper command-line arguments
55         if (args.length == 0) {
56            System.out.println("Usage: WeatherApp <city1> <city2> ...");
57            System.exit(1);
58         }
59
60         // simulate a server doing thread-per-request handling
61         try (var executor = Executors.newVirtualThreadPerTaskExecutor()) {
62            for (String city : args) {
63               executor.submit(
64                  // set the ScopedValue's value then run a task; the task
65                  // and its subtasks will have access to the ScopedValue
66                  () -> ScopedValue.where(CITY, city).run(
67                     // task: get and print the weather info for the city
68                     () -> {
69                        try {
70                           System.out.printf("%n%nWeather for %s:\n%s\n",
71                              city, getWeather());
72                        }
73                        catch (Exception ex) {
74                           System.err.printf(
75                              "Error fetching weather for %s: %s\n",
76                              city, ex.getMessage());
77                        }
78                     } // end nested lambda expression
79                  ) // end call to ScopedValue's run method
80               ); // end call to the executor object's submit method
81            } // end for loop
82         } // end try-with-resources statement
83      }
84
```

new StructuredTaskScope.ShutdownOnFailure()

```
85      // use structured concurrency to request current weather and forecast
86      public static String getWeather() throws Exception {
87         // URL encode the city parameter
88         String encodedCity =
89            URLEncoder.encode(CITY.get(), StandardCharsets.UTF_8);
90
91         // construct API URLs
92         String currentWeatherUrl = String.format(
93            "https://api.openweathermap.org/data/2.5/weather?q=%s" +
94            "&units=metric&appid=%s", encodedCity, API_KEY);
95         String forecastUrl = String.format(
96            "https://api.openweathermap.org/data/2.5/forecast?q=%s" +
97            "&units=metric&appid=%s", encodedCity, API_KEY);
98
99         // use StructuredTaskScope to manage concurrent tasks
100        try (var scope = new StructuredTaskScope.ShutdownOnFailure()) {
101           var currentWeatherTask = scope.fork(
102              () -> requestData(currentWeatherUrl, WeatherData.class));
103           var forecastTask = scope.fork(
104              () -> requestData(forecastUrl, ForecastResponse.class));
105
106           scope.join(); // waits for responses
107           scope.throwIfFailed(); // throws an exception if a task fails
108
109           // tasks completed successfully, retrieve results
110           WeatherData weatherData = currentWeatherTask.get();
111           ForecastResponse forecastResponse = forecastTask.get();
112
113           // format and return output
114           return formatWeatherOutput(weatherData, forecastResponse);
115        }
116     }
117
```

```
    // generic method to retrieve JSON data from OpenWeatherMap API
    private static <T> T requestData(String url, Class<T> type)
        throws Exception {
        // build the request to the OpenWeatherMap web service
        HttpRequest request = HttpRequest.newBuilder()
            .uri(URI.create(url)).GET().build();

        // send the HTTP request and get the response
        HttpResponse<String> response = HTTP_CLIENT.send(request,
            HttpResponse.BodyHandlers.ofString());

        // check if the request was successful; otherwise, throw Exception
        if (response.statusCode() != 200) {
            throw new Exception(String.format(
                "Error: HTTP status %d\n", response.statusCode()));
        }

        // use Jackson library to deserialize the JSON response
        // into the format specified by type
        return new ObjectMapper().readValue(response.body(), type);
    }
```

```
    // formats weather output for display
    private static String formatWeatherOutput(
        WeatherData weatherData, ForecastResponse forecastResponse) {

        // current temperature
        double tempC = weatherData.main().temp();
        double tempF = (tempC * 9 / 5) + 32;

        // date formatter
        DateTimeFormatter dateFormatter =
            DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm")
                .withZone(ZoneId.ofOffset("UTC",
                    ZoneOffset.ofTotalSeconds(weatherData.timezone())));

        // 7-day forecast of weather descriptions with min/max temperatures
        String forecastInfo = forecastResponse.list().stream().limit(16)
            .map(f -> {
                double tempCForecast = f.main().temp();
                double tempFForecast = (tempCForecast * 9 / 5) + 32;
                return String.format(
                    "%s: %s; temp: %.1f C (%.1f F)",
                    dateFormatter.format(Instant.ofEpochSecond(f.dt())),
                    f.weather()[0].main(), tempCForecast, tempFForecast);
            })
            .collect(Collectors.joining("\n"));

        return """
            %s Weather: %s
            Temperature: %.1f C (%.1f F)

            Forecast:
            %s""".formatted(weatherData.name(),
                weatherData.weather()[0].description(),
                tempC, tempF, forecastInfo);
    }
}
```

```
Weather for New Orleans:
New Orleans Weather: scattered clouds
Temperature: 13.6 C (56.5 F)

Forecast:
2025-03-06 21:00: Clouds; temp: 13.3 C (56.0 F)
2025-03-07 00:00: Clouds; temp: 12.3 C (54.2 F)
2025-03-07 03:00: Clouds; temp: 11.5 C (52.8 F)
2025-03-07 06:00: Clouds; temp: 11.6 C (52.8 F)
2025-03-07 09:00: Clouds; temp: 17.3 C (63.1 F)
2025-03-07 12:00: Clouds; temp: 22.7 C (72.8 F)
...
Weather for Los Angeles:
Los Angeles Weather: moderate rain
Temperature: 8.9 C (47.9 F)

Forecast:
2025-03-06 19:00: Rain; temp: 9.0 C (48.2 F)
2025-03-06 22:00: Clouds; temp: 9.4 C (48.9 F)
2025-03-07 01:00: Clear; temp: 9.3 C (48.7 F)
2025-03-07 04:00: Clear; temp: 8.4 C (47.1 F)
2025-03-07 07:00: Clear; temp: 8.7 C (47.6 F)
2025-03-07 10:00: Clear; temp: 12.8 C (55.0 F)
...


Weather for Boston:
Boston Weather: moderate rain
Temperature: 6.0 C (42.9 F)

Forecast:
2025-03-06 22:00: Clouds; temp: 4.9 C (40.8 F)
2025-03-07 01:00: Clouds; temp: 1.4 C (34.5 F)
2025-03-07 04:00: Clouds; temp: -2.9 C (26.8 F)
2025-03-07 07:00: Clouds; temp: -2.0 C (28.3 F)
2025-03-07 10:00: Clouds; temp: 1.7 C (35.1 F)
2025-03-07 13:00: Clouds; temp: 4.7 C (40.5 F)
...
```

```java
// Fig. 18.8: Buffer.java
// Buffer interface specifies methods called by Producer and Consumer.
public interface Buffer {
   // place int value into Buffer
   public void blockingPut(int value) throws InterruptedException;

   // return int value from Buffer
   public int blockingGet() throws InterruptedException;
}
```

```java
// Fig. 18.9: BlockingBuffer.java
// Creating a synchronized buffer using an ArrayBlockingQueue.
import java.util.concurrent.ArrayBlockingQueue;

public class BlockingBuffer implements Buffer {
   // BlockingBuffer uses an ArrayBlockingQueue to manage synchronization
   private final ArrayBlockingQueue<Integer> buffer =
      new ArrayBlockingQueue<>(1);

   // place value into buffer
   @Override
   public void blockingPut(int value) throws InterruptedException {
      buffer.put(value); // place value in buffer
      System.out.printf("%s%2d\t%s%d%n", "Producer writes ", value,
         "Buffer cells occupied: ", buffer.size());
   }

   // return value from buffer
   @Override
   public int blockingGet() throws InterruptedException {
      int readValue = buffer.take(); // remove value from buffer
      System.out.printf("%s %2d\t%s%d%n", "Consumer reads ",
         readValue, "Buffer cells occupied: ", buffer.size());

      return readValue;
   }
}
```

```
// Fig. 18.10: BlockingBufferTest.java
// Two threads manipulating a blocking buffer that properly
// implements the producer/consumer relationship.
import java.util.concurrent.Executors;
import java.util.concurrent.ThreadLocalRandom;

public class BlockingBufferTest {
   public static void main(String[] args) throws InterruptedException {
      // virtual-thread-per-task executor
      try (var executor = Executors.newVirtualThreadPerTaskExecutor()) {
         // create BlockingBuffer to store ints
         final Buffer buffer = new BlockingBuffer();

         // producer task writes 1-10 into buffer
         executor.execute(() -> {
            int sum = 0;

            for (int count = 1; count <= 10; count++) {
               try {
                  Thread.sleep(ThreadLocalRandom.current().nextInt(3000));
                  buffer.blockingPut(count); // write value into buffer
                  sum += count; // keep running total of values written
               }
               catch (InterruptedException exception) {
                  Thread.currentThread().interrupt();
               }
            }

            System.out.printf(
               "Producer done: Produced values totaling %d%n", sum);
         });

         // consumer task reads values from buffer
         executor.execute(() -> {
            int sum = 0;

            for (int count = 1; count <= 10; count++) {
               try {
                  Thread.sleep(ThreadLocalRandom.current().nextInt(3000));
                  sum += buffer.blockingGet(); // read value from buffer
               }
               catch (InterruptedException exception) {
                  Thread.currentThread().interrupt();
               }
            }

            System.out.printf(
               "%nConsumer done: Read values totaling %d%n", sum);
         });
      }
   }
}
```

```
Producer writes  1      Buffer cells occupied: 1
Consumer reads   1      Buffer cells occupied: 0
Producer writes  2      Buffer cells occupied: 1
Consumer reads   2      Buffer cells occupied: 0
Producer writes  3      Buffer cells occupied: 1
Consumer reads   3      Buffer cells occupied: 0
Producer writes  4      Buffer cells occupied: 1
Consumer reads   4      Buffer cells occupied: 0
Producer writes  5      Buffer cells occupied: 1
Consumer reads   5      Buffer cells occupied: 0
Producer writes  6      Buffer cells occupied: 1
Consumer reads   6      Buffer cells occupied: 0
Producer writes  7      Buffer cells occupied: 1
Consumer reads   7      Buffer cells occupied: 0
Producer writes  8      Buffer cells occupied: 1
Consumer reads   8      Buffer cells occupied: 0
Producer writes  9      Buffer cells occupied: 1
Consumer reads   9      Buffer cells occupied: 0
Producer writes 10      Buffer cells occupied: 1
Producer done: Produced values totaling 55
Consumer reads  10      Buffer cells occupied: 0

Consumer done: Read values totaling 55
```

https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/util/concurrent/package-summary.html

```java
// Fig. 18.11: FibonacciTask.java
// Task subclass for calculating Fibonacci numbers in the background
import javafx.concurrent.Task;

public class FibonacciTask extends Task<Long> {
   private final int n; // Fibonacci number to calculate

   // constructor
   public FibonacciTask(int n) {
      this.n = n;
   }

   // long-running code to be run in a worker thread
   @Override
   protected Long call() {
      updateMessage("Calculating...");
      long result = fibonacci(n);
      updateMessage("Done calculating.");
      return result;
   }

   // calculates nth Fibonacci number (Section 22.4 explains this method)
   public long fibonacci(long number) {
      if (number == 0 || number == 1) {
         return number;
      }
      else {
         return fibonacci(number - 1) + fibonacci(number - 2);
      }
   }
}
```

```
// Fig. 18.12: FibonacciNumbersController.java
// Using a Task to perform a long calculation
// outside the JavaFX application thread.
import java.util.concurrent.Executors;

import javafx.event.ActionEvent;
import javafx.fxml.FXML;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;

public class FibonacciNumbersController {
   @FXML private TextField numberTextField;
   @FXML private Button goButton;
   @FXML private Label messageLabel;
   @FXML private Label fibonacciLabel;
   @FXML private Label nthLabel;
   @FXML private Label nthFibonacciLabel;

   private long n1 = 0; // initialize with Fibonacci of 0
   private long n2 = 1; // initialize with Fibonacci of 1
   private int number = 1; // current Fibonacci number to display

```

```
   // starts FibonacciTask to calculate in background
   @FXML
   void goButtonPressed(ActionEvent event) {
      // get Fibonacci number to calculate
      try {
         int input = Integer.parseInt(numberTextField.getText());

         // create, configure and launch FibonacciTask
         var task = new FibonacciTask(input);

         // display task's messages in messageLabel
         messageLabel.textProperty().bind(task.messageProperty());

         // clear fibonacciLabel when task starts
         task.setOnRunning((succeededEvent) -> {
            goButton.setDisable(true);
            fibonacciLabel.setText("");
         });

         // set fibonacciLabel when task completes successfully
         task.setOnSucceeded((succeededEvent) -> {
            fibonacciLabel.setText(task.getValue().toString());
            goButton.setDisable(false);
         });

         // launch the FibonacciTask
         var executor = Executors.newFixedThreadPool(1);
         executor.execute(task); // start the task
         executor.shutdown(); // prevent more tasks from being scheduled
      }
      catch (NumberFormatException e) {
         numberTextField.setText("Enter an integer");
         numberTextField.selectAll();
         numberTextField.requestFocus();
      }
   }

```

```
   // calculates next Fibonacci value
   @FXML
   void nextNumberButtonPressed(ActionEvent event) {
      // display the next Fibonacci number
      nthLabel.setText("Fibonacci of " + number + ": ");
      nthFibonacciLabel.setText(String.valueOf(n2));
      long temp = n1 + n2;
      n1 = n2;
      n2 = temp;
      ++number;
   }
}
```

```java
// Fig. 18.13: PrimeCalculatorTask.java
// Calculates the first n primes, publishing them as they are found.
import java.util.Arrays;
import javafx.concurrent.Task;

public class PrimeCalculatorTask extends Task<Integer> {
   private final boolean[] primes; // boolean array for finding primes

   // constructor
   public PrimeCalculatorTask(int max) {
      primes = new boolean[max];
      Arrays.fill(primes, true); // initialize all primes elements to true
   }

   // long-running code to be run in a worker thread
   @Override
   protected Integer call() {
      int count = 0; // the number of primes found

      // starting at index 2 (the first prime number), cycle through and
      // set to false elements with indices that are multiples of i
      for (int i = 2; i < primes.length; i++) {
         if (isCancelled()) { // if calculation has been canceled
            updateMessage("Cancelled");
            return 0;


         }
         else {
            try {
               Thread.sleep(10); // slow the thread
            }
            catch (InterruptedException ex) {
               updateMessage("Interrupted");
               return 0;
            }

            updateProgress(i + 1, primes.length);

            if (primes[i]) { // i is prime
               ++count;
               updateMessage(String.format("Found %d primes", count));
               updateValue(i); // intermediate result

               // eliminate multiples of i
               for (int j = i + i; j < primes.length; j += i) {
                  primes[j] = false; // i is not prime
               }
            }
         }
      }

      return 0;
   }
}
```

```
1  // Fig. 18.14: FindPrimesController.java
2  // Displaying prime numbers as they're calculated; updating a ProgressBar
3  import java.util.concurrent.Executors;
4
5  import javafx.collections.FXCollections;
6  import javafx.collections.ObservableList;
7  import javafx.event.ActionEvent;
8  import javafx.fxml.FXML;
9  import javafx.scene.control.Button;
10 import javafx.scene.control.Label;
11 import javafx.scene.control.ListView;
12 import javafx.scene.control.ProgressBar;
13 import javafx.scene.control.TextField;
14
15 public class FindPrimesController {
16    @FXML private TextField inputTextField;
17    @FXML private Button getPrimesButton;
18    @FXML private ListView<Integer> primesListView;
19    @FXML private Button cancelButton;
20    @FXML private ProgressBar progressBar;
21    @FXML private Label statusLabel;
22
23    // stores the list of primes received from PrimeCalculatorTask
24    private ObservableList<Integer> primes =
25       FXCollections.observableArrayList();
26    private PrimeCalculatorTask task; // finds prime numbers
27
28    // binds primesListView's items to the ObservableList primes
29    @FXML
30    public void initialize() {
31       primesListView.setItems(primes);
32    }
33
```

```
34    // start calculating primes in the background
35    @FXML
36    void getPrimesButtonPressed(ActionEvent event) {
37       primes.clear();
38
39       // get Fibonacci number to calculate
40       try {
41          int input = Integer.parseInt(inputTextField.getText());
42          task = new PrimeCalculatorTask(input); // create task
43
44          // display task's messages in statusLabel
45          statusLabel.textProperty().bind(task.messageProperty());
46
47          // update progressBar based on task's progressProperty
48          progressBar.progressProperty().bind(task.progressProperty());
49
50          // store intermediate results in the ObservableList primes
51          task.valueProperty().addListener(
52             (observable, oldValue, newValue) -> {
53                if (newValue != 0) { // task returns 0 when it terminates
54                   primes.add(newValue);
55                   primesListView.scrollTo(
56                      primesListView.getItems().size());
57                }
58             });
59
```

```
60            // when task begins,
61            // disable getPrimesButton and enable cancelButton
62            task.setOnRunning((succeededEvent) -> {
63               getPrimesButton.setDisable(true);
64               cancelButton.setDisable(false);
65            });
66
67            // when task completes successfully,
68            // enable getPrimesButton and disable cancelButton
69            task.setOnSucceeded((succeededEvent) -> {
70               getPrimesButton.setDisable(false);
71               cancelButton.setDisable(true);
72            });
73
74            // launch the PrimeCalculatorTask
75            var executor = Executors.newVirtualThreadPerTaskExecutor();
76            executor.execute(task); // start the task
77            executor.shutdown();
78         }
79         catch (NumberFormatException e) {
80            inputTextField.setText("Enter an integer");
81            inputTextField.selectAll();
82            inputTextField.requestFocus();
83         }
84      }
85
```

```
86      // cancel task when user presses Cancel Button
87      @FXML
88      void cancelButtonPressed(ActionEvent event) {
89         if (task != null) {
90            task.cancel(); // terminate the task
91            getPrimesButton.setDisable(false);
92            cancelButton.setDisable(true);
93         }
94      }
95   }
```

https://platform.openai.com/settings/organization/billing/overview

https://platform.openai.com/signup

export OPENAI_API_KEY="*YourAPIKey*"

https://github.com/sashirestela/simple-openai

javac -cp .:../libs/OpenAIUtilities.jar *SourceFile*

java -cp .:../libs/OpenAIUtilities.jar *ClassName*

```
1  // Fig. 19.1: TextSummarizationDemo.java
2  // Summarizing a transcript as an abstract paragraph and key points.
3  import deitel.openai.OpenAIUtilities;
4  import deitel.openai.OpenAIUtilities.Message;
5
6  import java.nio.file.Files;
7  import java.nio.file.Path;
8  import java.util.List;
9
10 public class TextSummarizationDemo {
11    public static void main(String[] args) throws Exception {
```

```
12       // load transcript.txt
13       Path transcriptPath = Path.of(System.getProperty("user.home"),
14          "Documents", "examples", "ch19", "resources", "transcript.txt");
15       String transcript = Files.readString(transcriptPath);
16
```

```
        // get a summary abstract with OpenAI's gpt-4o model
        System.out.println("CREATE A SUMMARY ABSTRACT OF A TRANSCRIPT");
        String summaryAbstract = OpenAIUtilities.chat("gpt-4o",
           List.of(
              new Message("system", """
                 Given a transcript of a technical presentation, create a
                 concise, clear summary abstract in paragraph form, written
                 in a direct style that avoids prepositional phrases and
                 uses straightforward sentence structures. Focus on the key
                 points without referring to the speaker. Capture the key
                 ideas, so a person can understand it without reading the
                 full transcript."""),
              new Message("user", transcript)
           )
        );
        System.out.printf("%s%n%n", summaryAbstract);

```

```
CREATE A SUMMARY ABSTRACT OF A TRANSCRIPT
The "What's New in Java" lesson introduces updates and features from Java 10
to Java 22, with a brief mention of a Java 23 feature. These new elements are
integrated into various editions, including Java Fundamentals Third Edition
and others. The course is aligned with existing lessons to highlight how
these features can be applied, although placements in new products may vary.
The lesson covers core areas such as Java fundamentals, control statements,
data structures, and object-oriented programming. Key topics include func-
tional programming with lambdas and streams, JavaFX for graphical interfac-
es, and advanced concepts like concurrency and generative AI. The lesson also
explores practical tools like JShell for interactive development. Java ver-
sion-specific features are highlighted for developers operating on different
Java platforms, with an emphasis on long-term support releases. The new con-
tent will gradually replace older lessons as it is incorporated into updates.
```

```
public record Message(String role, String content) {}
```

https://platform.openai.com/docs/guides/prompt-engineering

```
        // get key points with OpenAI's gpt-4o model
        System.out.println("EXTRACT KEY POINTS FROM A TRANSCRIPT");
        String keyPoints = OpenAIUtilities.chat("gpt-4o",
           List.of(
              new Message("system", """
                 Given a transcript of a technical presentation, identify
                 the top 5 key points and present them as a numbered list.
                 For each point, use a concise, direct, clear, and
                 straightforward sentence. Avoid prepositional phrases."""),
              new Message("user", transcript)
           )
        );
        System.out.printf("%s%n", keyPoints);
     }
  }
```

```
EXTRACT KEY POINTS FROM A TRANSCRIPT
1. The lesson covers new Java programming features from versions Java 10 to
Java 22 with a preview of Java 23.
2. Java's recent features include executing source files directly and simpli-
fied main methods for easier coding.
3. Java 14 introduced switch expressions with powerful pattern matching capa-
bilities.
4. Java 21's sequenced collections offer unified method naming and reverse
iteration for ease of use.
5. Sealed classes and interfaces in Java give hierarchy designers complete
control over subclassing and implementation.
```

```java
// Fig. 19.2: SentimentAnalysisDemo.java
// Analyzing the sentiment of a transcript.
import deitel.openai.OpenAIUtilities;
import deitel.openai.OpenAIUtilities.Message;

import java.nio.file.Files;
import java.nio.file.Path;
import java.util.List;

public class SentimentAnalysisDemo {
   public static void main(String[] args) throws Exception {
      // load transcript.txt
      Path transcriptPath = Path.of(System.getProperty("user.home"),
         "Documents", "examples", "ch19", "resources", "transcript.txt");
      String transcript = Files.readString(transcriptPath);

      // analyze text sentiment with OpenAI's gpt-4o model
      System.out.println("ANALYZE SENTIMENT");
      String sentiment = OpenAIUtilities.chat("gpt-4o",
         List.of(
            new Message("system", """
               You are an expert in sentiment analysis. Analyze the
               following presentation transcript and state whether
               the sentiment is positive, negative, or neutral.
               Explain your analysis."""),
            new Message("user", transcript)
         )
      );
      System.out.printf("%s%n%n", sentiment);
   }
}
```

```
ANALYZE SENTIMENT
The sentiment of the presentation transcript is **neutral**.

### Analysis:

1. **Objective and Informative**:
   - The content is highly objective and focused on providing information
about the new features and updates in the upcoming Java course.

   - There is a detailed breakdown of what will be covered in the lesson, in-
cluding the structure and individual topics from fundamentals to high-end
Java concepts.

2. **Lack of Personal Emotional Expression**:
   - The speaker does not express personal emotions or opinions that would
indicate a positive or negative sentiment.
   - The language is professional and focused on conveying the course de-
tails.

3. **Balanced Tone**:
   - The transcript maintains a balanced tone throughout, avoiding any emo-
tionally charged language.
   - There are no exclamatory remarks or emotionally connotative words which
could sway the sentiment in either a positive or negative direction.

4. **Constructive and Supportive Notes**:
   - The speaker provides guidance on how to reach out for any questions and
mentions the availability of updates and support, which is informative but
does not express a particularly strong emotional sentiment.

Conclusively, the presentation's purpose is to inform the audience about the
structure and content of the upcoming Java course updates, rather than to
evoke a particular emotional response. Therefore, it is best categorized as
**neutral**.
```

https://www.markdownguide.org/getting-started/

https://www.markdownguide.org/basic-syntax/

```java
// Fig. 19.3: DescribeImageDemo.java
// Getting accessibility descriptions of images.
import deitel.openai.OpenAIUtilities;
import java.nio.file.Path;

public class DescribeImageDemo {
   public static void main(String[] args) throws Exception {
      // location of resources folder containing sample images
      Path resourcesPath = Path.of(System.getProperty("user.home"),
         "Documents", "examples", "ch19", "resources");

      // get an accessible description of a diagram using gpt-4o
      System.out.println("ACCESSIBLE DESCRIPTION OF A DIAGRAM");
      String diagramDescription =
         OpenAIUtilities.describeImage("gpt-4o", """
            The attached image is a UML activity diagram showing the
            flow of control through a Java 'for' loop with UML
            notes for additional context.""",
            Path.of(resourcesPath.toString(), "ForLoop.png"));
      System.out.printf("%s%n%n", diagramDescription);
```

```
ACCESSIBLE DESCRIPTION OF A DIAGRAM
The image is a highly detailed diagram showing the flow of control within a
logical structure of a Java 'for' loop, using the Unified Modeling Language
(UML) Activity Diagram standards. This diagram visually represents the se-
quence and operations involved in executing a 'for' loop in Java, a common
programming language used for building applications.

### Visual Breakdown:
1. **Diagram Format:** The activity diagram is structured to use standard UML
elements such as nodes, activities, decision points, transitions, and annota-
tions for depicting the logical flow.

2. **For Loop Structure:**
   - **Initialization:** The diagram starts with an initial node, indicating
the start of the logical process. This initial node flows into an activity
node representing the 'for' loop's initialization section (e.g., int i = 0).

   - **Condition Checking:** The next activity depicts the condition that
must be met for the loop to execute. This is shown with a diamond-shaped de-
cision node. It checks if the loop condition is true (e.g., i < 10), indicat-
ing whether the loop should continue or terminate.

   - **Loop Body:** From the true branch of the decision node, the flow pro-
ceeds to an activity node representing the loop body where actions (e.g.,
logic or statements) are executed if the condition is satisfied. These ac-
tions are enclosed within a rectangular activity node.

   - **Iteration:** A control flow arrow then typically points from the loop
body back to a small circular node, symbolizing the iterative aspect of the
loop (e.g., i++), where loop control variables are updated.

   - **Re-checking Condition:** After updating, control returns to the deci-
sion node to re-evaluate the condition.

3. **End of Loop:** If the condition is no longer satisfied, the control flow
follows the false branch from the decision node, leading to a final node.
This terminal node represents the end of the loop operation.

4. **Annotations:** The diagram may include UML notes or text boxes that pro-
vide additional context or explanations at various stages of the loop for
clarity. These are depicted generally as small rectangles connected to the
relevant part of the diagram by dashed lines.

5. **Flow Arrows:** The flow of control is indicated with arrows that con-
nect nodes and activities, forming a clear path illustrating the iterative
process of the 'for' loop.

6. **Styling and Color:** The diagram would use distinct shapes for decision
nodes (diamond), activity nodes (rounded rectangles), and annotations to
avoid confusion and provide a clear visual means to follow the logic.

Overall, this diagram serves as a tool to understand the execution flow of a
'for' loop in Java, detailing the iterative process in a structured graphi-
cal format. Each component is meticulously laid out to ensure the entire
mechanism is visually and logically represented.
```

```
        // get an accessible description of a photo using gpt-4o
        System.out.println("ACCESSIBLE DESCRIPTION OF A PHOTO");
        String imageDescription =
           OpenAIUtilities.describeImage("gpt-4o", """
                 Describe in detail the attached photo of a beach scene,
                 including every object in the photo.""",
              Path.of(resourcesPath.toString(), "beach.jpg"));
        System.out.printf("%s%n%n", imageDescription);
     }
  }
```

```
ACCESSIBLE DESCRIPTION OF A PHOTO
The photo depicts a serene beach scene under a clear blue sky. Central to the
image is a solitary palm tree with green and yellow fronds, casting a shadow
on the white sandy beach below. Two lounge chairs are placed side by side
facing the ocean, with one having a blue towel draped over its back.

In the background, the turquoise ocean stretches towards the horizon, where a
few small boats and a sailboat can be seen. On the right side of the image,
there is a wooden pier extending into the water, with a rustic structure fea-
turing a thatched roof. The ocean is calm, with gentle waves lapping against
the shore.

Part of a thatched roof is visible at the top edge of the photo, suggesting a
beach hut or shaded area not fully in view. The overall scene conveys a
peaceful and idyllic beach setting, ideal for relaxation.
```

```
1  // Fig. 19.4: TextTranslationDemo.java
2  // Translating text among spoken languages.
3  import deitel.openai.OpenAIUtilities;
4
5  public class TextTranslationDemo {
6     public static void main(String[] args) throws Exception {
7        // translate text with OpenAI's gpt-4o model
8        System.out.println("TRANSLATION DEMO");
9        String english =
10          "Today was a beautiful day. Tomorrow looks like bad weather.";
11       System.out.printf("ORIGINAL: %s%n%n", english);
12
13       System.out.println("Translating English to Spanish...");
14       String spanish =
15          OpenAIUtilities.translate("gpt-4o", english, "Spanish");
16       System.out.printf("SPANISH: %s%n%n", spanish);
17
18       System.out.println("Translating English to Japanese...");
19       String japanese =
20          OpenAIUtilities.translate("gpt-4o", english, "Japanese");
21       System.out.printf("JAPANESE: %s%n%n", japanese);
22
23       System.out.println("Translating Spanish to English...");
24       String spanishToEnglish =
25          OpenAIUtilities.translate("gpt-4o", spanish, "English");
26       System.out.printf(
27          "SPANISH TO ENGLISH: %s%n%n", spanishToEnglish);
28
```

```
        System.out.println("Translating Japanese to English...");
        String japaneseToEnglish =
           OpenAIUtilities.translate("gpt-4o", japanese, "English");
        System.out.printf(
           "JAPANESE TO ENGLISH: %s%n", japaneseToEnglish);
     }
  }
```

```
TRANSLATION DEMO
ORIGINAL: Today was a beautiful day. Tomorrow looks like bad weather.

Translating English to Spanish...
SPANISH: Hoy fue un día hermoso. Mañana parece que habrá mal tiempo.

Translating English to Japanese...
JAPANESE: 今日は美しい一日でした。明日は悪天候のようです。

Translating Spanish to English...
SPANISH TO ENGLISH: Today was a beautiful day. Tomorrow it seems there will
be bad weather.

Translating Japanese to English...
JAPANESE TO ENGLISH: Today was a beautiful day. It seems like there will be
bad weather tomorrow.
```

```
// Fig. 19.5: CodeGenerationDemo.java
// Generating code for a die-rolling simulation.
import deitel.openai.OpenAIUtilities;
import deitel.openai.OpenAIUtilities.Message;
import java.nio.file.Files;
import java.nio.file.StandardOpenOption;
import java.nio.file.Path;
import java.util.List;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class CodeGenerationDemo {
   public static void main(String[] args) throws Exception {
      // path to store Java source-code that's generated
      Path codePath = Path.of(System.getProperty("user.home"),
         "Documents", "examples", "ch19", "resources", "outputs");

      // generate Java code with OpenAI's gpt-4o model
      System.out.println("GENERATE DIE ROLLING SIMULATION CODE");
      String response = OpenAIUtilities.chat("gpt-4o",
         List.of(
            new Message("system",
               "You are an expert programmer in Java's latest version."),
            new Message("user", """
               Create a Java class named RollDie that simulates rolling
               a die 600,000,000 times as quickly as possible. Summarize
               the frequencies and display them left-aligned in a
               two-column format with the column heads "Face" and
               "Frequency". Avoid loops. In the code, use 3-space indents
               and restrict each code line to 74 characters or fewer.""")


         )
      );
      System.out.printf("%s%n%n", response);

      // use regex to extract the code and write it to disk;
      // the Pattern extracts all the text between the delimiters
      // "```java" and "```"
      Pattern pattern =
         Pattern.compile("```java\\s+(.*?)```", Pattern.DOTALL);
      Matcher matcher = pattern.matcher(response);

      // if there was a match, write it into the file RollDie.java
      if (matcher.find()) {
         Files.writeString(
            Path.of(codePath.toString(), "RollDie.java"),
            matcher.group(1).trim(),
            StandardOpenOption.TRUNCATE_EXISTING);
      }
   }
}
```

```
{
   "entities": [
      {
         "text": "Tim Berners-Lee",
         "tag: "PERSON"
      },
      {
         "text": "CERN",
         "tag: "ORG"
      },
      ...
   ]
}
```

```java
// Fig. 19.6: NamedEntityRecognitionDemo.java
// Identifying named entities and obtaining them as structured outputs.
import deitel.openai.OpenAIUtilities;
import deitel.openai.OpenAIUtilities.Message;
import deitel.openai.OpenAIUtilities.NamedEntities;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.List;

public class NamedEntityRecognitionDemo {
   public static void main(String[] args) throws Exception {
      // load web.txt
      Path transcriptPath = Path.of(System.getProperty("user.home"),
         "Documents", "examples", "ch19", "resources", "web.txt");
      String text = Files.readString(transcriptPath);

      // create a request to obtain named entities
      NamedEntities results = OpenAIUtilities.chatWithStructuredOutput(
         "gpt-4o",
         List.of(
            new Message("system", """
               You are an expert in named entity recognition (NER) using
               the OntoNotes 5.0 NER tag set. Analyze the following text
               and return the named entities in the specified structured
               JSON format. Do not include any additional information in
               your JSON response."""),
            new Message("user", text)
         ),
         OpenAIUtilities.NamedEntities.class
      );
```

```
        // display source text
        System.out.printf("SOURCE TEXT:%n%s%n%n", text);

        // display named entities
        System.out.println("ENTITIES:");
        results.entities().forEach(entity ->
           System.out.printf("%s: %s%n", entity.text(), entity.tag()));
     }
  }
```

```
SOURCE TEXT:
The World Wide Web (simply called "the web") is a collection of hardware and
software associated with the Internet that allows computer users to locate
and view documents (with various combinations of text, graphics, animations,
audios and videos) on almost any subject. In 1989, Tim Berners-Lee of CERN
(the European Organization for Nuclear Research) began developing HyperText
Markup Language (HTML)—the technology for sharing information via
"hyperlinked" text documents. He also wrote communication protocols such as
HyperText Transfer Protocol (HTTP) to form the backbone of his new hypertext
information system, which he referred to as the World Wide Web. In 1994,
Berners-Lee founded the World Wide Web Consortium (W3C, https://www.w3.org),
which was devoted to developing web technologies. A key W3C goal is to make
the web universally accessible to everyone regardless of disabilities,
language or culture.

ENTITIES:
World Wide Web: ORG
Internet: ORG
1989: DATE
Tim Berners-Lee: PERSON
CERN: ORG
European Organization for Nuclear Research: ORG
HyperText Markup Language: TECH
HTML: TECH
HyperText Transfer Protocol: TECH
HTTP: TECH
World Wide Web: ORG
1994: DATE
Berners-Lee: PERSON
World Wide Web Consortium: ORG
W3C: ORG
```

```
// Fig. 19.7: SpeechToTextDemo.java
// Transcribing audio files to text.
import deitel.openai.OpenAIUtilities;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.StandardOpenOption;

public class SpeechToTextDemo {
   public static void main(String[] args) throws Exception {
      // get path to resources folder
      Path resourcesFolder = Path.of(System.getProperty("user.home"),
         "Documents", "examples", "ch19", "resources");

      // get path to audio file WhatsNewInJavaOverview.m4a
      Path audioPath = Path.of(resourcesFolder.toString(),
         "WhatsNewInJavaOverview.m4a");

      // convert speech to text with OpenAI's whisper-1 model
      System.out.println("Waiting for Transcription...");
      String transcript = OpenAIUtilities.speechToText(
         "whisper-1", audioPath.toString());
      System.out.printf("TRANSCRIPT:%n%s%n", transcript);

      // write the transcription to a file, overwriting it if it exists
      Path transcriptPath = Path.of(resourcesFolder.toString(),
         "outputs", "WhatsNewInJavaOverview.txt");
      Files.writeString(transcriptPath, transcript,
         StandardOpenOption.CREATE,
         StandardOpenOption.TRUNCATE_EXISTING);
   }
}
```

```
Waiting for Transcription...
TRANSCRIPT:
Welcome to the What's New in Java lesson. In this lesson I have about two and
a half hours of new video with new examples that will be appearing in our
forthcoming new editions of our Java suite of products and that includes the
new edition of this video course, Java Fundamentals Third Edition, Java How
to Program Twelfth Edition, our Java textbook, and Java for Programmers Fifth
Edition, our corresponding professional book as well.

Throughout these videos I'll be presenting various key Java programming fea-
tures from Java 10 through Java 22 and I'll have a slide on one Java 23 fea-
ture as well that's related to some of the other examples that I'll be
presenting to you here. These videos in the What's New lesson are targeted to
existing lessons in the current product where the features in question could
potentially be used. That does not necessarily reflect where they'll appear
in the new suite of products, but it's a relatively good guess as to where
they'll appear. And as we produce the new versions of these videos, we will
post the new lessons online and eventually this lesson will go away as we
meld the new features into the new presentation as well.
...
```

https://platform.openai.com/docs/guides/text-to-speech

```java
// Fig. 19.8: TextToSpeechDemo.java
// Synthesizing speech from text.
import deitel.openai.OpenAIUtilities;
import java.nio.file.Path;

public class TextToSpeechDemo {
   public static void main(String[] args) throws Exception {
      // get path to the resources folder
      Path outputPath = Path.of(System.getProperty("user.home"),
         "Documents", "examples", "ch19", "resources", "outputs");

      // synthesize speech from text using OpenAI's tts-1-hd model
      System.out.println("TEXT-TO-SPEECH DEMO");
      System.out.println("Synthesizing English speech...");
      String english =
         "Today was a beautiful day. Tomorrow looks like bad weather.";
      OpenAIUtilities.textToSpeech("tts-1-hd", english, "onyx",
         Path.of(outputPath.toString(), "english.mp3").toString());

      System.out.println("Synthesizing Spanish speech...");
      String spanish =
         "Hoy fue un día hermoso. Mañana parece que habrá mal tiempo.";
      OpenAIUtilities.textToSpeech("tts-1-hd", spanish, "onyx",
         Path.of(outputPath.toString(), "spanish.mp3").toString());

      System.out.println("Synthesizing Japanese speech...");
      String japanese = "今日は美しい日でした。明日は悪天候になりそうです。";
      OpenAIUtilities.textToSpeech("tts-1-hd", japanese, "onyx",
         Path.of(outputPath.toString(), "japanese.mp3").toString());
   }
}
```

```
TEXT-TO-SPEECH DEMO
Synthesizing English speech...
Wrote audio to english.mp3
Synthesizing Spanish speech...
Wrote audio to spanish.mp3
Synthesizing Japanese speech...
Wrote audio to japanese.mp3
```

https://platform.openai.com/docs/guides/text-to-speech/voice-options#voice-options

```java
// Fig. 19.9: ImageGenerationDemo.java
// Creating images from text prompts.
import deitel.openai.OpenAIUtilities;
import java.nio.file.Path;

public class ImageGenerationDemo {
   public static void main(String[] args) throws Exception {
      // get path to the resources folder
      Path path = Path.of(System.getProperty("user.home"),
         "Documents", "examples", "ch19", "resources", "outputs");

      // generate images with OpenAI image API and dall-e-3 model
      System.out.println("IMAGE-GENERATION DEMO");
      String prompt = """
         Havanese dog as a Japanese anime character
         in neon colors against a black background""";
      System.out.printf("Generating image for:%n%s%n", prompt);
      OpenAIUtilities.image("dall-e-3", prompt,
         Path.of(path.toString(), "HavaneseAnime.png"));

      prompt = "Havanese dog in the style of Vincent Van Gogh";
      System.out.printf("%nGenerating image for:%n%s%n", prompt);
      OpenAIUtilities.image("dall-e-3", prompt,
         Path.of(path.toString(), "HavaneseVanGogh.png"));

      prompt = "Havanese dog in the style of Leonardo DaVinci";
      System.out.printf("%nGenerating image for:%n%s%n", prompt);
      OpenAIUtilities.image("dall-e-3", prompt,
         Path.of(path.toString(), "HavaneseDaVinci.png"));
   }
}
```

```
IMAGE-GENERATION DEMO
Generating image for:
Havanese dog as a Japanese anime character
in neon colors against a black background
Wrote image to /Users/pauldeitel/Documents/examples/ch19/resources/Havanese-
Anime.png

Generating image for:
Havanese dog in the style of Vincent Van Gogh
Wrote image to /Users/pauldeitel/Documents/examples/ch19/resources/Havanese-
VanGogh.png

Generating image for:
Havanese dog in the style of Leonardo DaVinci
Wrote image to /Users/pauldeitel/Documents/examples/ch19/resources/Havanese-
DaVinci.png
```

```java
// Fig. 19.10: SpeechToVTTDemo.java
// Transcribing the audio track from a video and creating closed captions.
import deitel.openai.OpenAIUtilities;

import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.StandardOpenOption;

public class SpeechToVTTDemo {
   public static void main(String[] args) throws Exception {
      // get path to resources folder
      Path resourcesFolder = Path.of(System.getProperty("user.home"),
         "Documents", "examples", "ch19", "resources");

      // get path to audio file ImplicitClass.m4a
      Path audioPath =
         Path.of(resourcesFolder.toString(), "ImplicitClass.m4a");

      // convert speech to closed captions with OpenAI's whisper-1 model
      System.out.println(
         "Transcribing audio and creating VTT captions file...");
      String captions =
         OpenAIUtilities.speechToVTT("whisper-1", audioPath);
      System.out.printf("CAPTIONS:%n%s%n", captions);

      // output the captions to a .vtt file
      var captionsPath = Path.of(resourcesFolder.toString(),
         "outputs", "ImplicitClass.vtt");
      Files.writeString(captionsPath, captions.trim(),
         StandardOpenOption.CREATE,
         StandardOpenOption.TRUNCATE_EXISTING);
   }
}
```

```
CAPTIONS:

WEBVTT

00:00:00.000 --> 00:00:07.700
In this video, we'll take another look at the JEP 463 capabilities, and in
particular,

00:00:07.700 --> 00:00:12.200
we're going to demonstrate an implicitly declared class.

00:00:12.200 --> 00:00:16.760
So basically, we're going to have a source code file with no class defini-
tion at all

00:00:16.760 --> 00:00:17.760
in it.

...
```

https://labs.google/fx/tools/video-fx

https://platform.openai.com/docs/guides/moderation

https://openai.com/policies/usage-policies/

```java
// Fig. 19.11: ModerationDemo.java
// Using the OpenAI moderation API to check for offensive content.
import deitel.openai.OpenAIUtilities;
import java.util.Scanner;

public class ModerationDemo {
   public static void main(String[] args) {
      var input = new Scanner(System.in);

      System.out.print("Enter prompt (or "exit"): ");
      String prompt = input.nextLine();

      while (!prompt.toLowerCase().equals("exit")) {
         var result = OpenAIUtilities.checkPrompt(prompt);

         if (!result.getFlagged()) {
            System.out.println(
               "Prompt not flagged for offensive content");
         }
         else {
            var categories = result.getCategories();
            var scores = result.getCategoryScores();

            // display report
            System.out.println("Offensive content categories & scores:");
            System.out.printf("""
               harassment: %b %.2f; threatening: %b %.2f
               hate: %b %.2f; threatening: %b %.2f
               self-harm: %b %.2f; instructions: %b %.2f; intent: %b %.2f
               sexual: %b %.2f; minors: %b %.2f
               violence: %b %.2f; graphic: %b %.2f%n""",
               categories.getHarassment(), scores.getHarassment(),
               categories.getHarassmentThreatening(),
               scores.getHarassmentThreatening(),

               categories.getHate(), scores.getHate(),
               categories.getHateThreatening(),
               scores.getHateThreatening(),
               categories.getSelfHarm(), scores.getSelfHarm(),
               categories.getSelfHarmInstructions(),
               scores.getSelfHarmInstructions(),
               categories.getSelfHarmIntent(), scores.getSelfHarmIntent(),
               categories.getSexual(), scores.getSexual(),
               categories.getSexualMinors(), scores.getSexualMinors(),
               categories.getViolence(), scores.getViolence(),
               categories.getViolencGraphic(),
               scores.getViolencGraphic());
         }

         System.out.print("\nEnter prompt (exit to terminate): ");
         prompt = input.nextLine();
      }
   }
}
```

```
Enter prompt (exit to terminate): I would die for an ice cream sundae right now.
Prompt not flagged for offensive content

Enter prompt (exit to terminate): I'm going to make them pay.
Offensive content categories & scores:
harassment: false 0.25; threatening: false 0.33
hate: false 0.01; threatening: false 0.00
self-harm: false 0.00; instructions: false 0.00; intent: false 0.00
sexual: false 0.00; minors: false 0.00
violence: true 0.42; graphic: false 0.00

Enter prompt (exit to terminate): exit
```

I would die for an ice cream sundae right now.

harassment: false 0.25; threatening: false 0.33

https://docs.oracle.com/en/java/javase/23/docs//api/java.base/java/util/concurrent/CompletableFuture.html

https://www.baeldung.com/java-completablefuture

https://dzone.com/articles/20-examples-of-using-javas-completablefuture

```java
// Fig. 19.12: OpenAIUtilities.java
// Helper methods and record classes to used encapsulate calling
// the OpenAI APIs via the Simple-OpenAI Java library.
package deitel.openai;

import com.fasterxml.jackson.databind.ObjectMapper;
import io.github.sashirestela.openai.SimpleOpenAI;
import io.github.sashirestela.openai.common.ResponseFormat;
import io.github.sashirestela.openai.common.ResponseFormat.JsonSchema;
import io.github.sashirestela.openai.common.audio.Voice;
import io.github.sashirestela.openai.common.content.ContentPart;
import io.github.sashirestela.openai.domain.audio.AudioResponseFormat;
import io.github.sashirestela.openai.domain.audio.SpeechRequest;
import io.github.sashirestela.openai.domain.audio.Transcription;
import io.github.sashirestela.openai.domain.audio.TranscriptionRequest;
import io.github.sashirestela.openai.domain.chat.Chat;
import io.github.sashirestela.openai.domain.chat.ChatMessage;
import io.github.sashirestela.openai.domain.chat.ChatRequest;
import io.github.sashirestela.openai.domain.image.ImageRequest;
import io.github.sashirestela.openai.domain.moderation.Moderation;
import io.github.sashirestela.openai.domain.moderation.ModerationRequest;
import io.github.sashirestela.openai.support.Base64Util;

import java.io.FileOutputStream;
import java.net.URL;
import java.net.URI;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.nio.file.StandardCopyOption;
import java.util.List;
import java.util.stream.Collectors;

public class OpenAIUtilities {
```

```java
   // record class used to simplify passing messages containing a role
   // and corresponding content to this class's chat and translate methods
   public record Message(String role, String content) {};

```

```java
   // create a SimpleOpenAI client object
   private final static SimpleOpenAI openAI = SimpleOpenAI.builder()
      .apiKey(System.getenv("OPENAI_API_KEY")).build();

```

```java
   // perform a chat completion
   public static String chat(String chatModel, List<Message> messages) {
      // start building a ChatRequest
      var chatRequest = ChatRequest.builder().model(chatModel);

      // add messages to the ChatRequest
      for (var currentMessage : makeChatMessages(messages)) {
         chatRequest = chatRequest.message(currentMessage);
      }

      var request = chatRequest.build(); // complete building the request

      // make the ChatRequest and get the Chat response
      var chatResponse = openAI.chatCompletions().create(request).join();

      return chatResponse.firstContent(); // return the response text
   }

```

https://platform.openai.com/docs/api-reference/chat/create

```
61      // utility method to convert our Message record objects into
62      // SystemMessage, UserMessage and AssistantMessage objects
63      private static List<ChatMessage> makeChatMessages(
64         List<Message> messages) {
65         // convert each Message record to a ChatMessage subclass object
66         return messages.stream()
67            .map(m -> switch(m.role().toLowerCase()) {
68               case "system" ->
69                  ChatMessage.SystemMessage.of(m.content());
70               case "user" -> ChatMessage.UserMessage.of(m.content());
71               case "assistant" ->
72                  ChatMessage.AssistantMessage.of(m.content());
73               default -> throw new IllegalArgumentException(
74                  "invalid role: " + m.role());
75            })
76            .toList();
77      }
78
```

```
79      // perform a chat completion
80      public static String describeImage(
81         String chatModel, String prompt, Path imagePath) {
82         // start building a ChatRequest for accessible image descriptions
83         var chatRequest = ChatRequest.builder()
84            .model(chatModel)
85            .message(ChatMessage.SystemMessage.of("""
86               You are an expert at creating accessible descriptions
87               of images per the World Wide Web Consortium's (W3C's)
88               guidelines. Given an image, explain it in detail for
89               people who are blind or have low vision."""))
90            .message(ChatMessage.UserMessage.of(
91               List.of(
92                  ContentPart.ContentPartText.of(prompt),
93                  ContentPart.ContentPartImageUrl.of(
94                     ContentPart.ContentPartImageUrl.ImageUrl.of(
95                        Base64Util.encode(imagePath.toString(),
96                           Base64Util.MediaType.IMAGE))))))
97            .build();
98
99         // make the ChatRequest and get the Chat response
100        var chatResponse =
101           openAI.chatCompletions().create(chatRequest).join();
102
103        return chatResponse.firstContent(); // return the response text
104     }
105
```

"``*base64EncodedImageString*"

```
106     // perform a chat completion for translation
107     public static String translate(
108        String chatModel, String textToTranslate, String targetLanguage) {
109
110        // build a ChatRequest to translate a String
111        var chatRequest = ChatRequest.builder().model(chatModel)
112           .message(ChatMessage.SystemMessage.of(String.format("""
113              You are an expert in natural language translation.
114              Translate the following text into %s.""", targetLanguage)))
115           .message(ChatMessage.UserMessage.of(textToTranslate))
116           .build();
117
118        // create and submit the ChatRequest and wait for the Chat response
119        Chat response = openAI.chatCompletions().create(chatRequest).join();
120
121        return response.firstContent(); // return the translation
122     }
123
```

```
124     // record classes to define the structure of a JSON response
125     // containing results of named entity recognition
126     public record NamedEntity(String text, String tag) {}
127     public record NamedEntities(List<NamedEntity> entities) {}
128
```

OpenAIUtilities.NamedEntities.class

Class<OpenAIUtilities.NamedEntities>

```
129     // generic method that performs a chat completion and
130     // returns structured output
131     public static <T> T chatWithStructuredOutput(String chatModel,
132        List<Message> messages, Class<T> jsonStructure) throws Exception {
133        // set up JSON schema for structured output based on jsonStructure
134        var schema = ResponseFormat.jsonSchema(JsonSchema.builder()
135           .name(jsonStructure.getSimpleName()) // schema name
136           .schemaClass(jsonStructure) // schema Java class
137           .build()
138        );
139
140        // start building a ChatRequest that returns structured output
141        var chatRequest = ChatRequest.builder()
142           .model(chatModel)
143           .responseFormat(schema);
144
145        // add messages to the ChatRequest
146        for (var currentMessage : makeChatMessages(messages)) {
147           chatRequest = chatRequest.message(currentMessage);
148        }
149
150        var request = chatRequest.build(); // complete building the request
151
152        // make the ChatRequest and get the Chat response
153        Chat response = openAI.chatCompletions().create(request).join();
154
155        // ObjectMapper for deserializing JSON into a NamedEntities object
156        var mapper = new ObjectMapper();
157
158        // return the structured response
159        return mapper.readValue(response.firstContent(), jsonStructure);
160     }
161
```

```
162     // convert audio to text transcription
163     public static String speechToText(String model, String audioPath) {
164        // build a TranscriptionRequest
165        var audioRequest = TranscriptionRequest.builder()
166           .file(Paths.get(audioPath))
167           .model(model)
168           .build();
169
170        // submit TranscriptionRequest and wait for Transcription
171        Transcription transcription =
172           openAI.audios().transcribe(audioRequest).join();
173        return transcription.getText(); // return transcription text
174     }
175
```

```
176     // synthesize speech from text
177     public static void textToSpeech(String model, String text,
178        String voice, String filename) throws Exception {
179        // build a SpeechRequest
180        var speechRequest = SpeechRequest.builder()
181           .model(model)
182           .input(text)
183           .voice(Voice.valueOf(voice.toUpperCase()))
184           .build();
185
186        // submit SpeechRequest and wait for audio, then save audio to file
187        try (var audioStream = openAI.audios().speak(speechRequest).join();
188             var audioFile = new FileOutputStream(filename)) {
189           audioFile.write(audioStream.readAllBytes());
190           System.out.printf("Wrote audio to %s%n", filename);
191        }
192     }
193
```

```
    // generate an image with Dall-E
    public static void image(
        String model, String prompt, Path imagePath) throws Exception {
        // build an ImageRequest with the specified model and prompt
        var imageRequest = ImageRequest.builder()
            .model(model)
            .prompt(prompt)
            .quality(ImageRequest.Quality.HD)
            .build();

        // submit ImageRequest and wait for image
        var imageResponse = openAI.images().create(imageRequest).join();

        // get image's URL
        String urlString = imageResponse.getFirst().getUrl();
        URL url = URI.create(urlString).toURL();

        // save image to PNG file
        try (var inputStream = url.openStream()) {
            Files.copy(inputStream, imagePath,
                StandardCopyOption.REPLACE_EXISTING);
            System.out.printf("Wrote image to %s%n", imagePath);
        }
    }

```

```
    // convert audio track of a video into closed captions
    public static String speechToVTT(String model, Path audioPath) {
        // build a TranscriptionRequest that returns JSON containing the
        // transcription segments and their start/end times
        var audioRequest = TranscriptionRequest.builder()
            .file(audioPath)
            .model(model)
            .responseFormat(AudioResponseFormat.VERBOSE_JSON)
            .build();

        // submit TranscriptionRequest and wait for Transcription
        Transcription transcription =
            openAI.audios().transcribe(audioRequest).join();

        // get the transcript's segments
        List<Transcription.Segment> segments = transcription.getSegments();

        // create a String representation of the transcription's
        // segments with their start and end times
        String transcriptionText = segments.stream()
            .map(segment -> String.format("Start: %f, End: %f, Text: %s",
                segment.getStart(), segment.getEnd(), segment.getText()))
            .collect(Collectors.joining("\n"));

        // issue a chat request to convert the audio
        // segment transcriptions into VTT format
        var request = ChatRequest.builder()
            .model("gpt-4o")

            .message(ChatMessage.SystemMessage.of("""
                You are a subtitle formatting assistant. Take the following
                transcription data and format it into VTT (WebVTT) format.
                Timestamps must have the hh:mm:ss.mmm format required by
                WebVTT. Include the required WEBVTT header at beginning of
                the file. Output the VTT content only."""))
            .message(ChatMessage.UserMessage.of(transcriptionText))
            .build();

        // make the ChatRequest and get the response
        var response = openAI.chatCompletions().create(request).join();

        // remove markdown code-block delimiters (```) at beginning
        // and end of response and return VTT-formatted captions
        var captions = response.firstContent();
        return captions.substring(3, captions.length() - 3);
    }

```

```
    // send text to OpenAI moderation API to check for offensive content;
    public static Moderation.ModerationResult checkPrompt(String prompt) {
       // build a ModerationRequest
       var moderationRequest = ModerationRequest.builder()
          .model("omni-moderation-latest") // OpenAI recommended model
          .input(List.of(prompt))
          .build();

       // submit the request and return the result
       var result = openAI.moderations().create(moderationRequest).join();
       return result.getResults().getFirst();
    }
}
```

```
public record IndexEntry(String entry, List<Integer> pages) {}
public record IndexEntries(List<IndexEntry> entries) {}
```

https://gutenberg.org/policy/terms_of_use.html

There once was a Java buff, Lee,
Who wrestled with streams endlessly.
When the lambdas got wild,
They just sat back and smiled,
"At least it's not pointers in C!"

```
public record Translation(String language, String translation) {}
public record Translations(String originalText,
   String originalLanguage, List<Translation> translations) {}
```

https://openai.com/index/navigating-the-challenges-and-opportunities-of-synthetic-voices/

https://platform.openai.com/docs/guides/text-to-speech

https://ai.meta.com/research/movie-gen/

https://deepmind.google/technologies/veo/veo-2/

https://magenta.tensorflow.org

https://magenta.tensorflow.org/demos

```
public record PartOfSpeech(String text, String part) {}
public record PartsOfSpeech(List<PartOfSpeech> parts) {}
```

https://www.uscourts.gov/about-federal-courts/educational-resources/about-educational-outreach/activity-resources/oxford

https://github.com/xerial/sqlite-jdbc/releases

Department	Location
413	New Jersey
611	Orlando
642	Los Angeles

https://www.sqlite.org/download.html

sqlite-tools-win-x64-3470200.zip

```
Get-Content books.sql | sqlite3.exe books.db
```

```
SQLite version 3.47.2 2024-12-07 20:39:59
Enter ".help" for usage hints.
sqlite>
```

```
sqlite> SELECT * FROM authors;
id  first      last
--  ---------  ------
1   Paul       Deitel
2   Harvey     Deitel
3   Abbey      Deitel
4   Dan        Quirk
5   Alexander  Wald
```

```
sqlite> SELECT * FROM titles;
isbn        title                           edition  copyright
----------  ------------------------------  -------  ---------
0135404673  Intro to Python for CS and DS    1        2020
0135224330  Python for Programmers           1        2019
0132151006  Internet & WWW How to Program    5        2012
0137598351  Java How to Program              12       2025
0137398395  C How to Program                 9        2022
0133406954  Visual Basic 2012 How to Program 6        2014
0134601548  Visual C# How to Program         6        2017
0136151574  Visual C++ How to Program        2        2008
0136905692  C++20 for Programmers            2        2023
0138092427  C++ How to Program               11       2024
0134444302  Android How to Program           3        2017
0134289366  Android 6 for Programmers        3        2016
```

```
sqlite> SELECT * FROM author_ISBN LIMIT 5;
id  isbn
--  ----------
1   0134289366
2   0134289366
5   0134289366
1   0135404673
2   0135404673
```

```
sqlite> SELECT first, last FROM authors;
first      last
---------  ------
Paul       Deitel
Harvey     Deitel
Abbey      Deitel
Dan        Quirk
Alexander  Wald
```

```
sqlite> SELECT title, edition, copyright
   ...> FROM titles
   ...> WHERE copyright > '2019';
title                          edition  copyright
-----------------------------  -------  ---------
Intro to Python for CS and DS  1        2020
Java How to Program            12       2025
C How to Program               9        2022
C++20 for Programmers          2        2023
C++ How to Program             11       2024
```

```
sqlite> SELECT id, first, last
   ...> FROM authors
   ...> WHERE last LIKE 'D%';
id  first   last
--  ------  ------
1   Paul    Deitel
2   Harvey  Deitel
3   Abbey   Deitel
```

```
sqlite> SELECT id, first, last
   ...> FROM authors
   ...> WHERE first LIKE '_b%';
id  first  last
--  -----  ------
3   Abbey  Deitel
```

```
sqlite> SELECT title FROM titles ORDER BY title ASC;
title
------------------------------
Android 6 for Programmers
Android How to Program
C How to Program
C++ How to Program
C++20 for Programmers
Internet & WWW How to Program
Intro to Python for CS and DS
Java How to Program
Python for Programmers
Visual Basic 2012 How to Program
Visual C# How to Program
Visual C++ How to Program
```

```
sqlite> SELECT id, first, last
   ...> FROM authors
   ...> ORDER BY last, first;
id  first      last
--  ---------  ------
3   Abbey      Deitel
2   Harvey     Deitel
1   Paul       Deitel
4   Dan        Quirk
5   Alexander  Wald
sqlite> SELECT id, first, last
   ...> FROM authors
   ...> ORDER BY last DESC, first ASC;
id  first      last
--  ---------  ------
5   Alexander  Wald
4   Dan        Quirk
3   Abbey      Deitel
2   Harvey     Deitel
1   Paul       Deitel
```

```
sqlite> SELECT isbn, title, edition, copyright
   ...> FROM titles
   ...> WHERE title LIKE '%How to Program'
   ...> ORDER BY title;
isbn        title                              edition  copyright
----------  ---------------------------------  -------  ---------
0134444302  Android How to Program             3        2017
0137398395  C How to Program                   9        2022
0138092427  C++ How to Program                 11       2024
0132151006  Internet & WWW How to Program      5        2012
0137598351  Java How to Program                12       2025
0133406954  Visual Basic 2012 How to Program   6        2014
0134601548  Visual C# How to Program           6        2017
0136151574  Visual C++ How to Program          2        2008

sqlite> SELECT first, last, isbn
   ...> FROM authors
   ...> INNER JOIN author_ISBN ON authors.id = author_ISBN.id
   ...> ORDER BY last, first LIMIT 5;
first   last    isbn
------  ------  ----------
Abbey   Deitel  0132151006
Abbey   Deitel  0133406954
Harvey  Deitel  0134289366
Harvey  Deitel  0135404673
Harvey  Deitel  0132151006
```

```
sqlite> INSERT INTO authors (first, last)
   ...> VALUES ('Mei-ling', 'Liu');

sqlite> SELECT id, first, last FROM authors;
id  first      last
--  ---------  ------
1   Paul       Deitel
2   Harvey     Deitel
3   Abbey      Deitel
4   Dan        Quirk
5   Alexander  Wald
6   Mei-ling   Liu

sqlite> UPDATE authors SET last='Cai'
   ...> WHERE last='Liu' AND first='Mei-ling';

sqlite> SELECT id, first, last FROM authors;
id  first      last
--  ---------  ------
1   Paul       Deitel
2   Harvey     Deitel
3   Abbey      Deitel
4   Dan        Quirk
5   Alexander  Wald
6   Mei-ling   Cai

sqlite> DELETE FROM authors WHERE id=6;

sqlite> SELECT id, first, last FROM authors;
id  first      last
--  ---------  ------
1   Paul       Deitel
2   Harvey     Deitel
3   Abbey      Deitel
4   Dan        Quirk
5   Alexander  Wald
```

```
java -cp ".:../libs/sqlite-jdbc-3.47.1.0.jar" DisplayAuthors
```

```java
// Fig. 20.1: DisplayAuthors.java
// Displaying the contents of the authors table.
import java.nio.file.Path;
import java.sql.Connection;
import java.sql.Statement;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.ResultSetMetaData;
import java.sql.SQLException;

public class DisplayAuthors {
   public static void main(String args[]) {
      // books.db location
      Path dbPath = Path.of(System.getProperty("user.home"),
         "Documents", "examples", "ch20", "books.db");

      final String url = "jdbc:sqlite:" + dbPath.toUri();
      final String query = "SELECT id, first, last FROM authors";

      // use try-with-resources to connect to and query the database
      try (
         Connection connection = DriverManager.getConnection(url);
         Statement statement = connection.createStatement();
         ResultSet resultSet = statement.executeQuery(query)) {

         // get ResultSet's meta data
         ResultSetMetaData metaData = resultSet.getMetaData();
         int numberOfColumns = metaData.getColumnCount();

         System.out.printf("Authors Table of Books Database:%n%n");

         // display the names of the columns in the ResultSet
         for (int i = 1; i <= numberOfColumns; i++) {
            System.out.printf("%-11s", metaData.getColumnName(i));
         }
         System.out.println();


         // display query results
         while (resultSet.next()) {
            for (int i = 1; i <= numberOfColumns; i++) {
               System.out.printf("%-11s", resultSet.getObject(i));
            }
            System.out.println();
         }
      }
      catch (SQLException sqlException) {
         sqlException.printStackTrace();
      }
   }
}
```

```
Authors Table of Books Database:

id         first      last
1          Paul       Deitel
2          Harvey     Deitel
3          Abbey      Deitel
4          Dan        Quirk
5          Alexander  Wald
```

```
javac --module-path $PATH_TO_FX \
   --add-modules javafx.controls,javafx.fxml *.java

javac --module-path $env:PATH_TO_FX `
   --add-modules javafx.controls,javafx.fxml *.java

javac --module-path %PATH_TO_FX% ^
   --add-modules javafx.controls,javafx.fxml *.java

java --module-path $PATH_TO_FX \
   --add-modules javafx.controls,javafx.fxml \
   -cp ".:../libs/sqlite-jdbc-3.47.1.0.jar" DisplayQueryResults
```

```
java --module-path $env:PATH_TO_FX `
   --add-modules javafx.controls,javafx.fxml `
   -cp ".;..\libs\sqlite-jdbc-3.47.1.0.jar" DisplayQueryResults

java --module-path %PATH_TO_FX% ^
   --add-modules javafx.controls,javafx.fxml ^
   -cp ".;..\libs\sqlite-jdbc-3.47.1.0.jar" DisplayQueryResults
```

```
1  // Fig. 20.2: DisplayQueryResultsController.java
2  // Controller for the DisplayQueryResults app
3  import java.sql.Connection;
4  import java.sql.DriverManager;
5  import java.sql.ResultSet;
6  import java.sql.ResultSetMetaData;
7  import java.sql.SQLException;
8  import java.sql.Statement;
9  import java.nio.file.Path;
10 import javafx.beans.property.SimpleObjectProperty;
11 import javafx.collections.FXCollections;
12 import javafx.collections.ObservableList;
13 import javafx.event.ActionEvent;
14 import javafx.fxml.FXML;
15 import javafx.scene.control.Alert;
16 import javafx.scene.control.Alert.AlertType;
17 import javafx.scene.control.TableColumn;
18 import javafx.scene.control.TableView;
19 import javafx.scene.control.TextArea;
20
```

```
21 public class DisplayQueryResultsController {
22    @FXML private TextArea queryTextArea;
23    @FXML private TableView<ObservableList<Object>> tableView;
24
25    // books.db location
26    private static final String DATABASE_URL = "jdbc:sqlite:" +
27       Path.of(System.getProperty("user.home"),
28          "Documents", "examples", "ch20", "books.db");
29
30    // default query retrieves all data from authors table
31    private static final String DEFAULT_QUERY = "SELECT * FROM authors";
32
33    private Connection connection;
34
```

```
35    // display DEFAULT_QUERY, connect to books.db and execute DEFAULT_QUERY
36    @FXML
37    public void initialize() {
38       queryTextArea.setText(DEFAULT_QUERY);
39
40       try {
41          connection = DriverManager.getConnection(DATABASE_URL);
42          executeQuery(DEFAULT_QUERY);
43       }
44       catch (SQLException sqlException) {
45          displayAlert(AlertType.ERROR, "Database Error",
46             sqlException.getMessage());
47          System.exit(1); // terminate application
48       }
49    }
50
```

```
51    // query the database and display results in TableView
52    @FXML
53    void submitQueryButtonPressed(ActionEvent event) {
54       executeQuery(queryTextArea.getText());
55    }
56
```

```
57      // performs the query and updates the ObservableList<Object>
58      // that the TableView uses to populate its rows and columns
59      private void executeQuery(String query) {
60         try (Statement statement = connection.createStatement();
61              ResultSet resultSet = statement.executeQuery(query)) {
62            tableView.getColumns().clear(); // remove current content
63
64            // create two-dimensional ObservableList for current ResultSet
65            ObservableList<ObservableList<Object>> data =
66               FXCollections.observableArrayList();
67
68            // use the ResultSetMetaData to get the column heads
69            // and set up the TableView's columns
70            ResultSetMetaData metaData = resultSet.getMetaData();
71            int columnCount = metaData.getColumnCount();
72
73            for (int i = 1; i <= columnCount; i++) {
74               // create a new TableColumn for each column in the ResultSet
75               var column = new TableColumn<ObservableList<Object>, Object>(
76                  metaData.getColumnName(i));
77
78               // map the column to the corresponding index in each row
79               final int columnIndex = i - 1;
80               column.setCellValueFactory(param ->
81                  new SimpleObjectProperty<>(
82                     param.getValue().get(columnIndex)));
83
84               // autoscale column width to divide space equally
85               column.prefWidthProperty().bind(
86                  tableView.widthProperty().divide(columnCount));
87
88               // add the column to the TableView
89               tableView.getColumns().add(column);
90            }
91
92            // populate the rows from the ResultSet
93            while (resultSet.next()) {
94               // create a new ObservableList to represent the row
95               ObservableList<Object> row =
96                  FXCollections.observableArrayList();
97
98               // populate the row with data from the ResultSet
99               for (int i = 1; i <= columnCount; i++) {
100                 row.add(resultSet.getObject(i));
101              }
102
103              // add the row to the data list
104              data.add(row);
105           }
106
107           // set the TableView's items to the ObservableList data
108           tableView.setItems(data);
109        }
110        catch (SQLException sqlException) {
111           displayAlert(AlertType.ERROR, "Query Error",
112              sqlException.getMessage());
113        }
114     }
115
```

```
116     // display an Alert dialog
117     private void displayAlert(
118        AlertType type, String title, String message) {
119        Alert alert = new Alert(type);
120        alert.setTitle(title);
121        alert.setContentText(message);
122        alert.showAndWait();
123     }
124  }
```

```
PreparedStatement authorBooks = connection.prepareStatement("""
   SELECT last, first, title FROM authors
   INNER JOIN author_ISBN ON authors.id=author_ISBN.id
   INNER JOIN titles ON titles.isbn=author_ISBN.isbn
   WHERE last = ? AND first = ?""");
```

```
authorBooks.setString(1, "Deitel");
authorBooks.setString(2, "Paul");
```

```
SELECT last, first, title FROM authors
INNER JOIN author_ISBN ON authors.id=author_ISBN.id
INNER JOIN Titles ON titles.isbn=author_ISBN.isbn
WHERE last = 'Deitel' AND first = 'Paul'
```

```
authorBooks.setString(1, "O'Brien");
```

```
sqlite3 addressbook.db < addressbook.sql
```

```
Get-Content addressbook.sql | sqlite3.exe addressbook.db
```

```
1  // Fig. 20.3: Person.java
2  // Person record class that represents an entry in an address book.
3  public record Person (
4     int id, String first, String last, String email, String phone) {
5
6     // custom String representation
7     @Override
8     public String toString() {
9        return String.format("%s, %s", last(), first());
10    }
11 }
```

```
1  // Fig. 20.4: PersonQueries.java
2  // PreparedStatements used by the AddressBook program.
3  import java.nio.file.Path;
4  import java.sql.Connection;
5  import java.sql.DriverManager;
6  import java.sql.PreparedStatement;
7  import iava.sql.ResultSet:
8  import java.sql.SQLException;
9  import java.util.List;
10 import java.util.ArrayList;
11
12 public class PersonQueries {
13    // addressbook.db location
14    private static final String DATABASE_URL = "jdbc:sqlite:" +
15       Path.of(System.getProperty("user.home"), "Documents",
16          "examples", "ch20", "fig20_03-05", "addressbook.db");
17
18    private Connection connection; // manages connection
19    private PreparedStatement selectAllPeople;
20    private PreparedStatement selectPeopleByLastName;
21    private PreparedStatement insertNewPerson;
22
```

```
23      // constructor
24      public PersonQueries() {
25         try {
26            connection = DriverManager.getConnection(DATABASE_URL);
27
28            // create query that selects all entries in the AddressBook
29            selectAllPeople = connection.prepareStatement(
30               "SELECT * FROM addresses ORDER BY last, first");
31
32            // create query that selects entries with last names
33            // that begin with the specified characters
34            selectPeopleByLastName = connection.prepareStatement("""
35               SELECT * FROM addresses WHERE last LIKE ?
36               ORDER BY last, first""");
37
38            // create insert operation that adds a new entry to the database
39            insertNewPerson = connection.prepareStatement("""
40               INSERT INTO addresses (first, last, email, phone)
41               VALUES (?, ?, ?, ?)""");
42         }
43         catch (SQLException sqlException) {
44            sqlException.printStackTrace();
45            System.exit(1);
46         }
47      }
48
```

```
49      // select all of the addresses in the database
50      public List<Person> getAllPeople() {
51         // executeQuery returns ResultSet containing matching entries
52         try (ResultSet resultSet = selectAllPeople.executeQuery()) {
53            List<Person> results = new ArrayList<>();
54
55            while (resultSet.next()) {
56               results.add(new Person(
57                  resultSet.getInt("id"),
58                  resultSet.getString("first"),
59                  resultSet.getString("last"),
60                  resultSet.getString("email"),
61                  resultSet.getString("phone")));
62            }
63
64            return results;
65         }
66         catch (SQLException sqlException) {
67            sqlException.printStackTrace();
68         }
69
70         return null;
71      }
72
```

```
73      // select person by last name
74      public List<Person> getPeopleByLastName(String lastName) {
75         try {
76            selectPeopleByLastName.setString(1, lastName); // set last name
77         }
78         catch (SQLException sqlException) {
79            sqlException.printStackTrace();
80            return null;
81         }
82
83         // executeQuery returns ResultSet containing matching entries
84         try (ResultSet resultSet = selectPeopleByLastName.executeQuery()) {
85            List<Person> results = new ArrayList<>();
86
87            while (resultSet.next()) {
88               results.add(new Person(
89                  resultSet.getInt("id"),
90                  resultSet.getString("first"),
91                  resultSet.getString("last"),
92                  resultSet.getString("email"),
93                  resultSet.getString("phone")));
94            }
95
96            return results;
97         }
98         catch (SQLException sqlException) {
99            sqlException.printStackTrace();
100           return null;
101        }
102     }
103
```

```
104     // add an entry
105     public int addPerson(String first, String last,
106        String email, String phone) {
107
108        // insert the new entry; returns # of rows updated
109        try {
110           // set parameters
111           insertNewPerson.setString(1, first);
112           insertNewPerson.setString(2, last);
113           insertNewPerson.setString(3, email);
114           insertNewPerson.setString(4, phone);
115
116           return insertNewPerson.executeUpdate();
117        }
118        catch (SQLException sqlException) {
119           sqlException.printStackTrace();
120           return 0;
121        }
122     }
123
124     // close the database connection
125     public void close() {
126        try {
127           connection.close();
128        }
129        catch (SQLException sqlException) {
130           sqlException.printStackTrace();
131        }
132     }
133  }
```

```
1  // Fig. 20.5: AddressBookController.java
2  // Controller for the AddressBook app
3  import java.util.List;
4
5  import javafx.collections.FXCollections;
6  import javafx.collections.ObservableList;
7  import javafx.event.ActionEvent;
8  import javafx.fxml.FXML;
9  import javafx.scene.control.Alert;
10 import javafx.scene.control.Alert.AlertType;
11 import javafx.scene.control.ListView;
12 import javafx.scene.control.TextField;
13
14 public class AddressBookController {
15    @FXML private ListView<Person> listView; // displays contact names
16    @FXML private TextField firstNameTextField;
17    @FXML private TextField lastNameTextField;
18    @FXML private TextField emailTextField;
19    @FXML private TextField phoneTextField;
20    @FXML private TextField findByLastNameTextField;
21
```

```
22    // interacts with the database
23    private final PersonQueries personQueries = new PersonQueries();
24
25    // stores list of Person objects that results from a database query
26    private final ObservableList<Person> contactList =
27       FXCollections.observableArrayList();
28
```

```
29    // populate listView and set up listener for selection events
30    public void initialize() {
31       // when ListView selection changes, display selected person's data
32       listView.getSelectionModel().selectedItemProperty().addListener(
33          (observableValue, oldValue, newValue) -> {
34             displayContact(newValue);
35          }
36       );
37
38       listView.setItems(contactList); // bind to contactsList
39       getAllEntries(); // populates contactList, which updates listView
40    }
41
```

```
42    // get all the entries from the database to populate contactList
43    private void getAllEntries() {
44       contactList.setAll(personQueries.getAllPeople());
45       selectFirstEntry();
46    }
47
48    // select first item in listView
49    private void selectFirstEntry() {
50       listView.getSelectionModel().selectFirst();
51    }
52
```

```
53    // display contact information
54    private void displayContact(Person person) {
55       if (person != null) {
56          firstNameTextField.setText(person.first());
57          lastNameTextField.setText(person.last());
58          emailTextField.setText(person.email());
59          phoneTextField.setText(person.phone());
60       }
61       else {
62          firstNameTextField.clear();
63          lastNameTextField.clear();
64          emailTextField.clear();
65          phoneTextField.clear();
66       }
67    }
68
```

```
69      // add a new entry
70      @FXML
71      void addEntryButtonPressed(ActionEvent event) {
72         int result = personQueries.addPerson(
73            firstNameTextField.getText(), lastNameTextField.getText(),
74            emailTextField.getText(), phoneTextField.getText());
75
76         if (result == 1) {
77            displayAlert(AlertType.INFORMATION, "Entry Added",
78               "New entry successfully added.");
79         }
80         else {
81            displayAlert(AlertType.ERROR, "Entry Not Added",
82               "Unable to add entry.");
83         }
84
85         getAllEntries();
86      }
87
```

```
88      // find entries with the specified last name
89      @FXML
90      void findButtonPressed(ActionEvent event) {
91         List<Person> people = personQueries.getPeopleByLastName(
92            findByLastNameTextField.getText() + "%");
93
94         if (people.size() > 0) { // display all entries
95            contactList.setAll(people);
96            selectFirstEntry();
97         }
98         else {
99            displayAlert(AlertType.INFORMATION, "Lastname Not Found",
100              "There are no entries with the specified last name.");
101        }
102     }
103
```

```
104     // browse all the entries
105     @FXML
106     void browseAllButtonPressed(ActionEvent event) {
107        getAllEntries();
108     }
109
```

```
110     // display an Alert dialog
111     private void displayAlert(
112        AlertType type, String title, String message) {
113        var alert = new Alert(type);
114        alert.setTitle(title);
115        alert.setContentText(message);
116        alert.showAndWait();
117     }
118  }
```

java.base@23	jdk.compiler@23
java.compiler@23	jdk.crypto.cryptoki@23
java.datatransfer@23	**jdk.crypto.ec@23**
java.desktop@23	**jdk.crypto.mscapi@23**
java.instrument@23	jdk.dynalink@23
java.logging@23	jdk.editpad@23
java.management@23	**jdk.graal.compiler@23**
java.management.rmi@23	**jdk.graal.compiler.management@23**
java.naming@23	jdk.hotspot.agent@23
java.net.http@23	jdk.httpserver@23
java.prefs@23	jdk.incubator.vector@23
java.rmi@23	**jdk.internal.ed@23**
java.scripting@23	**jdk.internal.jvmstat@23**
java.se@23	**jdk.internal.le@23**
java.security.jgss@23	**jdk.internal.md@23**
java.security.sasl@23	**jdk.internal.opt@23**
java.smartcardio@23	**jdk.internal.vm.ci@23**
java.sql@23	jdk.jartool@23
java.sql.rowset@23	jdk.javadoc@23
java.transaction.xa@23	jdk.jcmd@23
java.xml@23	jdk.jconsole@23
java.xml.crypto@23	jdk.jdeps@23
jdk.accessibility@23	jdk.jdi@23
jdk.attach@23	jdk.jdwp.agent@23
jdk.charsets@23	jdk.jfr@23
jdk.jlink@23	jdk.naming.rmi@23
jdk.jpackage@23	jdk.net@23
jdk.jshell@23	jdk.nio.mapmode@23
jdk.jsobject@23	jdk.sctp@23
jdk.jstatd@23	jdk.security.auth@23
jdk.localedata@23	jdk.security.jgss@23
jdk.management@23	**jdk.unsupported@23**
jdk.management.agent@23	**jdk.unsupported.desktop@23**
jdk.management.jfr@23	jdk.xml.dom@23
jdk.naming.dns@23	jdk.zipfs@23

https://docs.oracle.com/en/java/javase/23/docs/api/index.html

`requires transitive` *modulename*;

`opens` *package* `to` *comma-separated-list-of-modules*

https://docs.oracle.com/javase/tutorial/reflect/

AppFolder

src

ModuleNameFolder

PackageFolders

JavaSourceCodeFiles

module-info.java

com.deitel.sample.firstpackage;
com.deitel.sample.secondpackage;
com.deitel.sample.thirdpackage;

https://docs.oracle.com/en/java/javase/23/docs/api/java.base/module-summary.html

java --describe-module java.base

```
exports java.io
exports java.lang
exports java.lang.annotation
exports java.lang.classfile
exports java.lang.classfile.attribute
exports java.lang.classfile.components
exports java.lang.classfile.constantpool
exports java.lang.classfile.instruction
exports java.lang.constant
exports java.lang.foreign
exports java.lang.invoke
exports java.lang.module
exports java.lang.ref
exports java.lang.reflect
exports java.lang.runtime
exports java.math
exports java.net
exports java.net.spi
exports java.nio
exports java.nio.channels
exports java.nio.channels.spi
exports java.nio.charset
exports java.nio.charset.spi
exports java.nio.file
exports java.nio.file.attribute
exports java.nio.file.spi
exports java.security
exports java.security.cert
exports java.security.interfaces
exports java.security.spec
```

```
exports java.text
exports java.text.spi
exports java.time
exports java.time.chrono
exports java.time.format
exports java.time.temporal
exports java.time.zone
exports java.util
exports java.util.concurrent
exports java.util.concurrent.atomic
exports java.util.concurrent.locks
exports java.util.function
exports java.util.jar
exports java.util.random
exports java.util.regex
exports java.util.spi
exports java.util.stream
exports java.util.zip
exports javax.crypto
exports javax.crypto.interfaces
exports javax.crypto.spec
exports javax.net
exports javax.net.ssl
exports javax.security.auth
exports javax.security.auth.callback
exports javax.security.auth.login
exports javax.security.auth.spi
exports javax.security.auth.x500
exports javax.security.cert
```

```
...
uses java.util.spi.CalendarDataProvider
uses java.util.spi.CurrencyNameProvider
...
provides java.nio.file.spi.FileSystemProvider
   with jdk.internal.jrtfs.JrtFileSystemProvider
...
qualified exports com.sun.security.ntlm to java.security.sasl
...
contains jdk.internal
contains jdk.internal.access.foreign
...
```

uses java.util.spi.CalendarDataProvider

provides java.nio.file.spi.FileSystemProvider
with jdk.internal.jrtfs.JrtFileSystemProvider

qualified exports com.sun.security.ntlm to java.security.sasl

contains com.sun.crypto.provider

```
1  // Fig. 21.1: Welcome.java
2  // Welcome class that will be placed in a module
3  package com.deitel.welcome; // all classes in modules must be packaged
4
5  public class Welcome {
6     public static void main(String[] args) {
7        // class System is in package java.lang from the java.base module
8        System.out.println("Welcome to the Java Platform Module System!");
9     }
10 }
```

```
1  // Fig. 21.2: module-info.java
2  // Module declaration for the com.deitel.welcome module
3  module com.deitel.welcome {
4     requires java.base; // implicit in all modules, so can be omitted
5  }
```

```
javac -d mods/com.deitel.welcome \
   src/com.deitel.welcome/module-info.java \
   src/com.deitel.welcome/com/deitel/welcome/Welcome.java

java --module-path mods --describe-module com.deitel.welcome
```

```
com.deitel.welcome file:///Users/pauldeitel/Documents/examples/ch21/
WelcomeApp/mods/com.deitel.welcome/
requires java.base
contains com.deitel.welcome
```

```
java --module-path mods \
   --module com.deitel.welcome/com.deitel.welcome.Welcome
```

```
Welcome to the Java Platform Module System!
```

```
jar --create --file jars/com.deitel.welcome.jar \
   --main-class com.deitel.welcome.Welcome \
   -C mods/com.deitel.welcome .

jar -cfe jars/com.deitel.welcome.jar \
   com.deitel.welcome.Welcome \
   -C mods/com.deitel.welcome .

jar --describe-module --file jars/com.deitel.welcome.jar
```

```
com.deitel.welcome jar:file:///Users/pauldeitel/Documents/examples/ch21/
   WelcomeApp/jars/com.deitel.welcome.jar!/module-info.class
requires java.base
contains com.deitel.welcome
main-class com.deitel.welcome.Welcome
```

```
java --module-path jars -m com.deitel.welcome

java -p jars -m com.deitel.welcome

Welcome to the Java Platform Module System!
```

```
java --module-path jars \
   -m com.deitel.welcome/com.deitel.welcome.Welcome

java -p jars -m com.deitel.welcome/com.deitel.welcome.Welcome
```

```
// Fig. 21.3: Time1.java
// Class Time1 that will be placed in a module.
package com.deitel.timelibrary;

public class Time1 {
   private int hour; // 0 - 23
   private int minute; // 0 - 59
   private int second; // 0 - 59

   // set a new time value using universal time; throw an
   // exception if the hour, minute or second is invalid
   public void setTime(int hour, int minute, int second) {
      // validate hour, minute and second
      if (hour < 0 || hour >= 24 || minute < 0 || minute >= 60 ||
         second < 0 || second >= 60) {
         throw new IllegalArgumentException(
            "hour, minute and/or second was out of range");
      }

      this.hour = hour;
      this.minute = minute;
      this.second = second;
   }

   // convert to String in universal-time format (HH:MM:SS)
   public String toUniversalString() {
      return String.format("%02d:%02d:%02d", hour, minute, second);
   }

   // convert to String in standard-time format (H:MM:SS AM or PM)
   public String toString() {
      return String.format("%d:%02d:%02d %s",
         ((hour == 0 || hour == 12) ? 12 : hour % 12),
         minute, second, (hour < 12 ? "AM" : "PM"));
   }
}
```

```
// Fig. 21.4: module-info.java
// Module declaration for the com.deitel.timelibrary module
module com.deitel.timelibrary {
   exports com.deitel.timelibrary; // package available to other modules
}
```

```java
// Fig. 21.5: Time1Test.java
// Time1 object used in an app.
package com.deitel.timetest;

import com.deitel.timelibrary.Time1;

public class Time1Test {
   public static void main(String[] args) {
      // create and initialize a Time1 object
      var time = new Time1(); // invokes Time1 constructor

      // output string representations of the time
      displayTime("After time object is created", time);
      System.out.println();

      // change time and output updated time
      time.setTime(13, 27, 6);
      displayTime("After calling setTime", time);
      System.out.println();

      // attempt to set time with invalid values
      try {
         time.setTime(99, 99, 99); // all values out of range
      }
      catch (IllegalArgumentException e) {
         System.out.printf("Exception: %s%n%n", e.getMessage());
      }

      // display time after attempt to set invalid values
      displayTime("After calling setTime with invalid values", time);
   }

   // displays a Time1 object in 24-hour and 12-hour formats
   private static void displayTime(String header, Time1 t) {
      System.out.printf("%s%nUniversal time: %s%nStandard time: %s%n",
         header, t.toUniversalString(), t.toString());
   }
}
```

```java
// Fig. 21.6: module-info.java
// Module declaration for the com.deitel.timetest module
module com.deitel.timetest {
   requires com.deitel.timelibrary;
}
```

```
javac -d mods/com.deitel.timelibrary \
   src/com.deitel.timelibrary/module-info.java \
   src/com.deitel.timelibrary/com/deitel/timelibrary/Time1.java

javac --module-path mods -d mods/com.deitel.timetest \
   src/com.deitel.timetest/module-info.java \
   src/com.deitel.timetest/com/deitel/timetest/Time1Test.java

java --module-path mods \
   -m com.deitel.timetest/com.deitel.timetest.Time1Test
```

```
After time object is created
Universal time: 00:00:00
Standard time: 12:00:00 AM

After calling setTime
Universal time: 13:27:06
Standard time: 1:27:06 PM

Exception: hour, minute and/or second was out of range

After calling setTime with invalid values
Universal time: 13:27:06
Standard time: 1:27:06 PM
```

```
jar --create -f jars/com.deitel.timelibrary.jar \
   -C mods/com.deitel.timelibrary .
```

```
jar --create -f jars/com.deitel.timetest.jar \
   --main-class com.deitel.timetest.Time1Test \
   -C mods/com.deitel.timetest .

java --module-path jars -m com.deitel.timetest
```

```
src/com.deitel.timetest/com/deitel/timetest/Time1Test.java:5: error: package
com.deitel.timelibrary is not visible
import com.deitel.timelibrary.Time1;
                ^
  (package com.deitel.timelibrary is declared in module com.deitel.timeli-
brary, which does not export it)
1 error
```

```
requires transitive java.logging;
requires transitive java.transaction.xa;
requires transitive java.xml;
```

```
module-info.java:2: error: cyclic dependence involving mymodule
   requires mymodule;
            ^
1 error
```

```
javac --module-source-path src \
   --module-path mods -d mods \
   src/module1/module-info.java \
   src/module1/package1/Class1.java \
   src/module2/module-info.java \
   src/module2/package2/Class2.java
```

```
src/module1/module-info.java:9: error: cyclic dependence involving module2
   requires module2;
            ^
src/module2/module-info.java:9: error: cyclic dependence involving module1
   requires module1;
            ^
2 errors
```

```
jdeps --jdk-internals YourClassName.class

jdeps --jdk-internals YourJARName.jar

jdeps --module-path jars -m com.deitel.welcome
```

```
com.deitel.welcome
 [file:///Users/pauldeitel/Documents/examples/ch21/WelcomeApp/jars/
   com.deitel.welcome.jar]
   requires java.base (@23)
com.deitel.welcome -> java.base
   com.deitel.welcome -> java.io        java.base
   com.deitel.welcome -> java.lang      java.base
```

jdeps jars/com.deitel.welcome.jar

```
jdeps mods/com.deitel.welcome/com/deitel/welcome/Welcome.class
```

```
Welcome.class -> java.base
   com.deitel.welcome -> java.io        java.base
   com.deitel.welcome -> java.lang      java.base
```

jdeps -v jars/com.deitel.welcome.jar

```
com.deitel.welcome
 [file:///Users/pauldeitel/Documents/examples/ch21/WelcomeApp/jars/
   com.deitel.welcome.jar]
   requires java.base (@23)
com.deitel.welcome -> java.base
   com.deitel.welcome.Welcome -> java.io.PrintStream      java.base
   com.deitel.welcome.Welcome -> java.lang.Object         java.base
   com.deitel.welcome.Welcome -> java.lang.String         java.base
   com.deitel.welcome.Welcome -> java.lang.System         java.base
```

jdeps --dot-output . jars/com.deitel.welcome.jar

```
dot -Tpng summary.dot -Gdpi=600 -Nfontname="Open Sans" \
   -Nshape=box -o WelcomeApp.png
```

https://docs.oracle.com/en/java/javase/23/docs/specs/man/jdeps.html

com.deitel.videoplayer.VideoPlayerController

package com.deitel.videoplayer;

package com.deitel.videoplayer;

```
1  // Fig. 21.8: module-info.java
2  // Module declaration for the com.deitel.videoplayer module
3  module com.deitel.videoplayer {
4     requires javafx.controls;
5     requires javafx.fxml;
6     requires javafx.media;
7     requires org.controlsfx.controls;
8
9     exports com.deitel.videoplayer;
10    opens com.deitel.videoplayer to javafx.fxml;
11 }
```

```
javac --module-path "$PATH_TO_FX":mods \
   -d mods/com.deitel.videoplayer \
   src/com.deitel.videoplayer/module-info.java \
   src/com.deitel.videoplayer/com/deitel/videoplayer/*.java

java --module-path "$PATH_TO_FX":mods \
   -m com.deitel.videoplayer/com.deitel.videoplayer.VideoPlayer
```

```
jdeps --module-path "$PATH_TO_FX":mods --dot-output . \
   --recursive --module com.deitel.videoplayer

dot -Tpng summary.dot -Gdpi=600 -Nfontname="Open Sans" \
   -Nshape=box -o VideoPlayer.png

jlink --module-path "$JAVA_HOME/jmods" --add-modules java.base \
   --output javabaseruntime

javabaseruntime/bin/java --module-path mods \
   --module com.deitel.welcome/com.deitel.welcome.Welcome
```

```
Welcome to the Java Platform Module System!
```

```
javabaseruntime/bin/java --list-modules

jlink --module-path "$JAVA_HOME/jmods" \
   --add-modules java.sql --output javasqlruntime

javasqlruntime/bin/java --list-modules

jlink --module-path jars:"$JAVA_HOME/jmods" \
   --add-modules com.deitel.welcome --output welcomeruntime

welcomeruntime/bin/java --list-modules

welcomeruntime/bin/java -m com.deitel.welcome
```

```
Welcome to the Java Platform Module System!
```

```
welcomeruntime/bin/java --show-module-resolution -m com.deitel.welcome
```

```
root com.deitel.welcome jrt:/com.deitel.welcome
Welcome to the Java Platform Module System!
```

```
root com.deitel.videoplayer file:///Users/pauldeitel/Documents/examples/ch21/
   VideoPlayer/mods/com.deitel.videoplayer/
com.deitel.videoplayer requires javafx.media file:///Users/pauldeitel/
   Downloads/javafx-sdk-23.0.1/lib/javafx.media.jar
com.deitel.videoplayer requires javafx.controls file:///Users/pauldeitel/
   Downloads/javafx-sdk-23.0.1/lib/javafx.controls.jar
com.deitel.videoplayer requires javafx.fxml file:///Users/pauldeitel/
   Downloads/javafx-sdk-23.0.1/lib/javafx.fxml.jar
com.deitel.videoplayer requires org.controlsfx.controls file:///Users/
   pauldeitel/Documents/examples/ch21/VideoPlayer/mods/controlsfx.jar
javafx.media requires javafx.base file:///Users/pauldeitel/Downloads/
   javafx-sdk-23.0.1/lib/javafx.base.jar
javafx.media requires javafx.graphics file:///Users/pauldeitel/Downloads/
   javafx-sdk-23.0.1/lib/javafx.graphics.jar
javafx.controls requires javafx.base file:///Users/pauldeitel/Downloads/
   javafx-sdk-23.0.1/lib/javafx.base.jar
javafx.controls requires javafx.graphics file:///Users/pauldeitel/Downloads/
   javafx-sdk-23.0.1/lib/javafx.graphics.jar
...
```

```java
// Fig. 21.9: ProblemProvider.java
// Service-provider interface for obtaining a Problem
package com.deitel.mathtutor.spi;

public interface ProblemProvider {
   public Problem getProblem();
}
```

```java
// Fig. 21.10: Problem.java
// Problem superclass that contains information about a math problem.
package com.deitel.mathtutor.spi;

public abstract class Problem {
   private int leftOperand;
   private int rightOperand;
   private int result;
   private String operation;

   // constructor
   public Problem(int leftOperand, int rightOperand, String operation) {
      this.leftOperand = leftOperand;
      this.rightOperand = rightOperand;
      this.operation = operation;
   }

   // gets the leftOperand
   public int getLeftOperand() {return leftOperand;}

   // gets the rightOperand
   public int getRightOperand() {return rightOperand;}

   // gets the operation
   public String getOperation() {return operation;}

   // gets the result
   public abstract int getResult();
}
```

```java
// Fig. 21.11: MathTutor.java
// Math tutoring app using ProblemProviders to display math problems.
package com.deitel.mathtutor;

import java.util.List;
import java.util.Scanner;
import java.util.ServiceLoader;
import java.util.ServiceLoader.Provider;
import java.util.random.RandomGenerator;
import java.util.stream.Collectors;
import com.deitel.mathtutor.spi.Problem;
import com.deitel.mathtutor.spi.ProblemProvider;

public class MathTutor {
   private static Scanner input = new Scanner(System.in);

   public static void main(String[] args) {
```

```java
      // get a service loader for ProblemProviders
      ServiceLoader<ProblemProvider> serviceLoader =
         ServiceLoader.load(ProblemProvider.class);

```

```java
      // get the list of service providers
      List<Provider<ProblemProvider>> providersList =
         serviceLoader.stream().collect(Collectors.toList());

      // check whether there are any providers
      if (providersList.isEmpty()) {
         System.out.println(
            "Terminating MathTutor: No problem providers found.");
         return;
      }

```

```
        boolean shouldContinue = true;
        var random = RandomGenerator.getDefault();

        do {
           // choose a ProblemProvider at random
           ProblemProvider provider =
              providersList.get(random.nextInt(providersList.size())).get();

           // get the Problem
           Problem problem = provider.getProblem();

           // display the problem to the user
           showProblem(problem);
        } while (playAgain());
     }

     // show the math problem to the user
     private static void showProblem(Problem problem) {
        var problemStatement = String.format("%nWhat is %d %s %d? ",
           problem.getLeftOperand(), problem.getOperation(),
           problem.getRightOperand());


        // display problem and get answer from user
        System.out.printf(problemStatement);
        int answer = input.nextInt();

        while (answer != problem.getResult()) {
           System.out.println("Incorrect. Please try again.");
           System.out.printf(problemStatement);
           answer = input.nextInt();
        }

        System.out.println("Correct!");
     }

     // play again?
     private static boolean playAgain() {
        System.out.printf("%nTry another? y to continue, n to terminate: ");
        String response = input.next();

        return response.toLowerCase().startsWith("y");
     }
  }
```

```
// Fig. 21.12: module-info.java
// Module declaration for the com.deitel.mathtutor module
module com.deitel.mathtutor {
   exports com.deitel.mathtutor.spi; // package for provider interface

   uses com.deitel.mathtutor.spi.ProblemProvider;
}
```

```
javac -d mods/com.deitel.mathtutor \
   src/com.deitel.mathtutor/module-info.java \
   src/com.deitel.mathtutor/com/deitel/mathtutor/MathTutor.java \
   src/com.deitel.mathtutor/com/deitel/mathtutor/spi/*.java
```

```
java --module-path mods/com.deitel.mathtutor \
   -m com.deitel.mathtutor/com.deitel.mathtutor.MathTutor
```

```
Terminating MathTutor: No problem providers found.
```

```
// Fig. 21.13: AdditionProblemProvider.java
// AdditionProblemProvider implementation of interface
// ProblemProvider for the MathTutor app.
package com.deitel.additionprovider;

import java.util.random.RandomGenerator;
import com.deitel.mathtutor.spi.Problem;
import com.deitel.mathtutor.spi.ProblemProvider;

public class AdditionProblemProvider implements ProblemProvider {
   private static RandomGenerator random = RandomGenerator.getDefault();

   // returns a new addition problem
   @Override
   public Problem getProblem() {
      return new Problem(random.nextInt(10), random.nextInt(10), "+") {
         // override getResult to add the operands
         @Override
         public int getResult() {
            return getLeftOperand() + getRightOperand();
         }
      };
   }
}
```

```
// Fig. 21.14: module-info.java
// Module declaration for the com.deitel.additionprovider module
module com.deitel.additionprovider {
   requires com.deitel.mathtutor;

   provides com.deitel.mathtutor.spi.ProblemProvider
      with com.deitel.additionprovider.AdditionProblemProvider;
}
```

```
javac --module-path mods -d mods/com.deitel.additionprovider \
   src/com.deitel.additionprovider/module-info.java \
   src/com.deitel.additionprovider/com/deitel/additionprovider/\
AdditionProblemProvider.java
```

```
java --module-path mods \
   -m com.deitel.mathtutor/com.deitel.mathtutor.MathTutor
```

```
What is 9 + 6? 15
Correct!

Try another? y to continue, n to terminate: y

What is 2 + 6? 7
Incorrect. Please try again.

What is 2 + 6? 8
Correct!

Try another? y to continue, n to terminate: n
```

```java
// Fig. 21.15: MultiplicationProblemProvider.java
// MultiplicationProblemProvider implementation of interface
// ProblemProvider for the MathTutor app.
package com.deitel.multiplicationprovider;

import java.util.random.RandomGenerator;
import com.deitel.mathtutor.spi.Problem;
import com.deitel.mathtutor.spi.ProblemProvider;

public class MultiplicationProblemProvider implements ProblemProvider {
   private static RandomGenerator random = RandomGenerator.getDefault();

   // returns a new multiplication problem
   @Override
   public Problem getProblem() {
      return new Problem(random.nextInt(10), random.nextInt(10), "*") {
         // override getResult to add the operands
         @Override
         public int getResult() {
            return getLeftOperand() * getRightOperand();
         }
      };
   }
}
```

```java
// Fig. 21.16: module-info.java
// Module declaration for the com.deitel.multiplicationprovider module
module com.deitel.multiplicationprovider {
   requires com.deitel.mathtutor;

   provides com.deitel.mathtutor.spi.ProblemProvider with
      com.deitel.multiplicationprovider.MultiplicationProblemProvider;
}
```

```
javac --module-path mods \
   -d mods/com.deitel.multiplicationprovider \
   src/com.deitel.multiplicationprovider/module-info.java \
   src/com.deitel.multiplicationprovider/com/deitel/\
multiplicationprovider/MultiplicationProblemProvider.java

java --module-path mods \
   -m com.deitel.mathtutor/com.deitel.mathtutor.MathTutor
```

```
What is 4 * 8? 20
Incorrect. Please try again.

What is 4 * 8? 32
Correct!

Try another? y to continue, n to terminate: y

What is 3 * 6? 18
Correct!

Try another? y to continue, n to terminate: y

What is 3 + 7? 10
Correct!

Try another? y to continue, n to terminate: y

What is 9 + 3? 12
Correct!

Try another? y to continue, n to terminate: n
```

```
factorial = 1;

for (int counter = number; counter >= 1; --counter) {
   factorial *= counter;
}
```

```
// Fig. 22.1: FactorialCalculator.java
// Recursive factorial method.
import java.math.BigInteger;

public class FactorialCalculator {
   // recursive method factorial (assumes its parameter is >= 0)
   public static BigInteger factorial(int number) {
      if (number <= 1) { // test for base case
         return BigInteger.ONE; // base cases: 0! = 1 and 1! = 1
      }
      else { // recursion step
         return BigInteger.valueOf(number).multiply(
            factorial(number - 1));
      }
   }

   public static void main(String[] args) {
      // calculate the factorials of 0 through 10
      for (int counter = 0; counter <= 10; ++counter) {
         System.out.printf("%d! = %d%n", counter, factorial(counter));
      }

      // display the factorials of 20, 30 and 40
      System.out.printf("%n20!: %s%n30!: %s%n40!: %s%n",
         factorial(20), factorial(30), factorial(40));
   }
}
```

```
0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800
```

```
20!: 2432902008176640000
30!: 265252859812191058636308480000000
40!: 815915283247897734345611269596115894272000000000
```

$$0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots$$

$$\text{fibonacci}(0) = 0$$

$$\text{fibonacci}(1) = 1$$

$$\text{fibonacci}(n) = \text{fibonacci}(n-1) + \text{fibonacci}(n-2)$$

```java
// Fig. 22.2: FibonacciCalculator.java
// Recursive fibonacci method.
import java.math.BigInteger;

public class FibonacciCalculator {
   private static BigInteger TWO = BigInteger.valueOf(2);

   // recursive declaration of method fibonacci
   public static BigInteger fibonacci(BigInteger number) {
      if (number.equals(BigInteger.ZERO) ||
          number.equals(BigInteger.ONE)) { // base cases
         return number;
      }
      else { // recursion step
         return fibonacci(number.subtract(BigInteger.ONE)).add(
            fibonacci(number.subtract(TWO)));
      }
   }

   public static void main(String[] args) {
      // displays the fibonacci values from 0-40
      for (int counter = 0; counter <= 40; ++counter) {
         System.out.printf("Fibonacci of %d is: %d%n", counter,
            fibonacci(BigInteger.valueOf(counter)));
      }
   }
}
```

```
Fibonacci of 0 is: 0
Fibonacci of 1 is: 1
Fibonacci of 2 is: 1
Fibonacci of 3 is: 2
Fibonacci of 4 is: 3
Fibonacci of 5 is: 5
Fibonacci of 6 is: 8
Fibonacci of 7 is: 13
Fibonacci of 8 is: 21
Fibonacci of 9 is: 34
Fibonacci of 10 is: 55
...
Fibonacci of 37 is: 24157817
Fibonacci of 38 is: 39088169
Fibonacci of 39 is: 63245986
Fibonacci of 40 is: 102334155
```

```java
// Fig. 22.3: FactorialCalculator.java
// Iterative factorial method.
import java.math.BigInteger;

public class FactorialCalculator {
   // iterative declaration of method factorial
   public static BigInteger factorial(int number) {
      var result = BigInteger.ONE;

      // iteratively calculate factorial
      for (int i = number; i >= 1; --i) {
         result = result.multiply(BigInteger.valueOf(i));
      }

      return result;
   }

   public static void main(String[] args) {
      // calculate the factorials of 0 through 10
      for (int counter = 0; counter <= 10; ++counter) {
         System.out.printf("%d! = %s%n", counter, factorial(counter));
      }

```

```
        // display the factorials of 20, 30 and 40
        System.out.printf("%n20!: %s%n30!: %s%n40!: %s%n",
           factorial(20), factorial(30), factorial(40));
     }
  }
```

```
0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800

20!: 2432902008176640000
30!: 265252859812191058636308480000000
40!: 815915283247897734345611269596115894272000000000
```

```
1  // Fig. 22.4: TowersOfHanoi.java
2  // Towers of Hanoi solution with a recursive method.
3  public class TowersOfHanoi {
4     // recursively move disks between towers
5     public static void solveTowers(int disks, int sourcePeg,
6        int destinationPeg, int tempPeg) {
7        // base case -- only one disk to move
8        if (disks == 1) {
9           System.out.printf("%n%d --> %d", sourcePeg, destinationPeg);
10          return;
11       }
12
13       // recursion step -- move (disk - 1) disks from sourcePeg
14       // to tempPeg using destinationPeg
15       solveTowers(disks - 1, sourcePeg, tempPeg, destinationPeg);
16
17       // move last disk from sourcePeg to destinationPeg
18       System.out.printf("%n%d --> %d", sourcePeg, destinationPeg);
19
20       // move (disks - 1) disks from tempPeg to destinationPeg
21       solveTowers(disks - 1, tempPeg, destinationPeg, sourcePeg);
22    }
23
24    public static void main(String[] args) {
25       int startPeg = 1; // value 1 used to indicate startPeg in output
26       int endPeg = 3; // value 3 used to indicate endPeg in output
27       int tempPeg = 2; // value 2 used to indicate tempPeg in output
28       int totalDisks = 3; // number of disks
29
30       // initial nonrecursive call: move all disks.
31       solveTowers(totalDisks, startPeg, endPeg, tempPeg);
32    }
33 }
```

```
1 --> 3
1 --> 2
3 --> 2
1 --> 3
2 --> 1
2 --> 3
1 --> 3
```

```
xD = xA + (xC - xA) / 2 - (yC - yA) / 2;
yD = yA + (yC - yA) / 2 + (xC - xA) / 2;
```

```java
// Fig. 22.5: FractalController.java
// Drawing the "Lo feather fractal" using recursion.
import javafx.event.ActionEvent;
import javafx.fxml.FXML;
import javafx.scene.canvas.Canvas;
import javafx.scene.canvas.GraphicsContext;
import javafx.scene.control.ColorPicker;
import javafx.scene.control.Label;
import javafx.scene.paint.Color;


public class FractalController {
   // constants
   private static final int MIN_LEVEL = 0;
   private static final int MAX_LEVEL = 15;

   // instance variables that refer to GUI components
   @FXML private Canvas canvas;
   @FXML private ColorPicker colorPicker;
   @FXML private Label levelLabel;


   // other instance variables
   private Color currentColor = Color.BLUE;
   private int level = MIN_LEVEL; // initial fractal level
   private GraphicsContext gc; // used to draw on Canvas

   // initialize the controller
   public void initialize() {
      levelLabel.setText("Level: " + level);
      colorPicker.setValue(currentColor); // start with blue
      gc = canvas.getGraphicsContext2D(); // get the GraphicsContext
      drawFractal();
   }

   // sets currentColor when user chooses a new Color
   @FXML
   void colorSelected(ActionEvent event) {
      currentColor = colorPicker.getValue(); // get new Color
      drawFractal();
   }

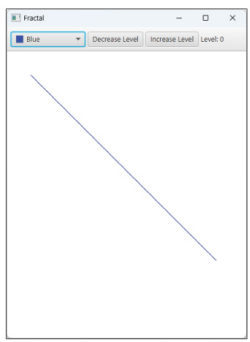
   // decrease level and redraw fractal
   @FXML
   void decreaseLevelButtonPressed(ActionEvent event) {
      if (level > MIN_LEVEL) {
         --level;
         levelLabel.setText("Level: " + level);
         drawFractal();
      }
   }

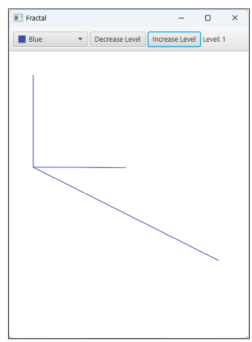

   // increase level and redraw fractal
   @FXML
   void increaseLevelButtonPressed(ActionEvent event) {
      if (level < MAX_LEVEL) {
         ++level;
         levelLabel.setText("Level: " + level);
         drawFractal();
      }
   }

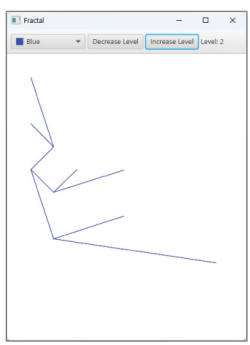
   // clear Canvas, set drawing color and draw the fractal
   private void drawFractal() {
      gc.clearRect(0, 0, canvas.getWidth(), canvas.getHeight());
      gc.setStroke(currentColor);
      drawFractal(level, 40, 40, 350, 350);
   }

   // draw fractal recursively
   public void drawFractal(int level, int xA, int yA, int xB, int yB) {
      // base case: draw a line connecting two given points
      if (level == 0) {
         gc.strokeLine(xA, yA, xB, yB);
      }
```

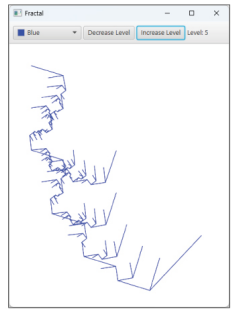
```
75          else { // recursion step: determine new points, draw next level
76             // calculate midpoint between (xA, yA) and (xB, yB)
77             int xC = (xA + xB) / 2;
78             int yC = (yA + yB) / 2;
79
80             // calculate the fourth point (xD, yD) which forms an
81             // isosceles right triangle between (xA, yA) and (xC, yC)
82             // where the right angle is at (xD, yD)
83             int xD = xA + (xC - xA) / 2 - (yC - yA) / 2;
84             int yD = yA + (yC - yA) / 2 + (xC - xA) / 2;
85
86             // recursively draw the Fractal
87             drawFractal(level - 1, xD, yD, xA, yA);
88             drawFractal(level - 1, xD, yD, xC, yC);
89             drawFractal(level - 1, xD, yD, xB, yB);
90          }
91       }
92    }
```

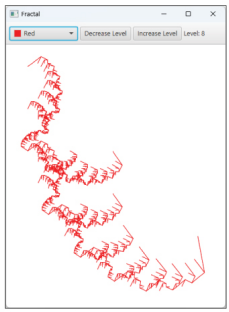


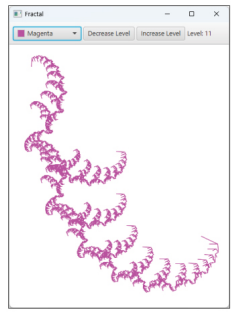




Fractal
Blue
Decrease Level
Increase Level
Level: 4


Fractal
Blue
Decrease Level
Increase Level
Level: 5


Fractal
Red
Decrease Level
Increase Level
Level: 8


Fractal
Magenta
Decrease Level
Increase Level
Level: 11

```
|  Welcome to JShell -- Version 23
|  For an introduction type: /help intro

jshell>
```

```java
// Fig. A.1: Welcome1.java
// Text-printing program.

public class Welcome1 {
   // main method begins program execution
   public static void main(String[] args) {
      System.out.println("Welcome to Java Programming!");
   } // end method main
} // end class Welcome1
```

```
Welcome to Java Programming!
```

```
jshell> System.out.println("Welcome to Java Programming!")
Welcome to Java Programming!

jshell>

jshell> System.out.print("Welcome to Java Programming!")
Welcome to Java Programming!
jshell>
```

```
jshell> int sum = number1 + number2
|  Error:
|  cannot find symbol
|    symbol:   variable number2
|  int sum = number1 + number2;
|                      ^-----^

jshell>

jshell> /history

System.out.println("Welcome to Java Programming!")
System.out.print("Welcome to Java Programming!")
int number1
int number1 = 30
number1 = 45
int sum = number1 + number2
/history

jshell>

jshell> int sum = number1 + number2
```

```
jshell> int sum = number1 + number2
sum ==> 117

jshell>
```

```
jshell> /list

   1 : System.out.println("Welcome to Java Programming!")
   2 : System.out.print("Welcome to Java Programming!")
   4 : int number1 = 30;
   5 : number1 = 45
   6 : int number2 = 72;
   7 : int sum = number1 + number2;

jshell>
```

```
jshell> /1
System.out.println("Welcome to Java Programming!")
Welcome to Java Programming!

jshell>
```

```
jshell> /!
System.out.println("Welcome to Java Programming!")
Welcome to Java Programming!

jshell>

System.out.println("Welcome to Java Programming!")
```

```
jshell> /list

   1 : System.out.println("Welcome to Java Programming!")
   2 : System.out.print("Welcome to Java Programming!")
   4 : int number1 = 30;
   5 : number1 = 45
   6 : int number2 = 72;
   7 : int sum = number1 + number2;
   8 : System.out.println("Welcome to Java Programming!")
   9 : System.out.println("Welcome to Java Programming!")

jshell>
```

```
jshell> /list

   1 : System.out.println("Welcome to Java Programming!")
   2 : System.out.print("Welcome to Java Programming!")
   4 : int number1 = 30;
   5 : number1 = 45
   6 : int number2 = 72;
   7 : int sum = number1 + number2;
   8 : System.out.println("Welcome to Java Programming!")
   9 : System.out.println("Welcome to Java Programming!")
  10 : 11 + 5

jshell>

jshell> /list

   1 : System.out.println("Welcome to Java Programming!")
   2 : System.out.print("Welcome to Java Programming!")
   4 : int number1 = 30;
   5 : number1 = 45
   6 : int number2 = 72;
   7 : int sum = number1 + number2;
   8 : System.out.println("Welcome to Java Programming!")
   9 : System.out.println("Welcome to Java Programming!")
  10 : 11 + 5
  11 : sum = number1 + $10

jshell>

jshell> if ($1 < $2) {
   ...>     System.out.printf("%d < %d%n", $1, $2);
   ...> }
45 < 72

jshell>

jshell> /list

   1 : 45
   2 : 72
   3 : if ($1 < $2) {
           System.out.printf("%d < %d%n", $1, $2);
       }

jshell>
```

```
if ($1 > $2) {
   System.out.printf("%d > %d%n", $1, $2);
}
```

```
jshell> /list

   1 : 45
   2 : 72
   3 : if ($1 < $2) {
          System.out.printf("%d < %d%n", $1, $2);
       }
   4 : if ($1 > $2) {
          System.out.printf("%d > %d%n", $1, $2);
       }
   5 : $1 = 100

jshell>
```

```
jshell> /4
if ($1 > $2) {
   System.out.printf("%d > %d%n", $1, $2);
}
100 > 72

jshell>
```

```
jshell> /list

   1 : 45
   2 : 72
   3 : if ($1 < $2) {
          System.out.printf("%d < %d%n", $1, $2);
       }
   4 : if ($1 > $2) {
          System.out.printf("%d > %d%n", $1, $2);
       }
   5 : $1 = 100
   6 : if ($1 > $2) {
          System.out.printf("%d > %d%n", $1, $2);
       }

jshell>
```

```
Scanner input = new Scanner(System.in);

System.out.print("Enter first integer: ");
int number1 = input.nextInt();
```

```
jshell> Scanner input = new Scanner(System.in)
input ==> java.util.Scanner[delimiters=\p{javaWhitespace}+] ...
\E][infinity string=\Q∞\E]

jshell>
```

```
jshell> System.out.print("Enter first integer: ")
Enter first integer:
jshell>

jshell> int number1 = input.nextInt()
_

jshell> int number1 = input.nextInt()
45
number1 ==> 45

jshell>

jshell> public class Account {
   ...>    private String name;
   ...>
   ...>    public void setName(String name) {
   ...>       this.name = name;
   ...>    }
   ...>
   ...>    public String getName() {
   ...>       return name;
   ...>    }
   ...> }
|  created class Account

jshell>

jshell> account = new Account()
account ==> Account@2d928643

jshell>

jshell> /vars
|    Account account = Account@2d928643
|    Account $4 = Account@49993335

jshell>

jshell> account.setName("Amanda")

jshell>
```

```
jshell> System.out.println(account.getName())
Amanda

jshell>
```

```
jshell> account.getName()
jshell> String _= account.getName()
```

```
jshell> String name = account.getName()
name ==> "Amanda"

jshell>
```

```
jshell> String greeting = "hello"
```

```
jshell> greeting.
charAt(                 chars()                 codePointAt(
codePointBefore(        codePointCount(         codePoints()
compareTo(              compareToIgnoreCase(    concat(
contains(               contentEquals(          describeConstable()
endsWith(               equals(                 equalsIgnoreCase(
formatted(              getBytes(               getChars(
getClass()              hashCode()              indent(
indexOf(                intern()                isBlank()
isEmpty()               lastIndexOf(            length()
lines()                 matches(                notify()
notifyAll()             offsetByCodePoints(     regionMatches(
repeat(                 replace(                replaceAll(
replaceFirst(           resolveConstantDesc(    split(
splitWithDelimiters(    startsWith(             strip()
stripIndent()           stripLeading()          stripTrailing()
subSequence(            substring(              toCharArray()
toLowerCase(            toString()              toUpperCase(
transform(              translateEscapes()      trim()
wait(

jshell> greeting.
```

```
jshell> greeting.endsWith("o")
$2 ==> true


jshell>
```

```
jshell> /
/!          /?          /drop       /edit       /env        /exit
/help       /history    /imports    /list       /methods    /open
/reload     /reset      /save       /set        /types      /vars

<press tab again to see synopsis>
jshell> /
```

```
jshell> /h
/help       /history

<press tab again to see synopsis>
jshell> /h
```

```
jshell> Math.
<press tab again to see all possible completions; total possible
completions: 103>
jshell> Math.
```

```
jshell> Math.
E                IEEEremainder(     PI                TAU
abs(             absExact(          acos(             addExact(
asin(            atan(              atan2(            cbrt(
ceil(            ceilDiv(           ceilDivExact(     ceilMod(
clamp(           class              copySign(         cos(
cosh(            decrementExact(    divideExact(      exp(
expm1(           floor(             floorDiv(         floorDivExact(
floorMod(        fma(               getExponent(      hypot(
incrementExact(  log(               log10(            log1p(
max(             min(               multiplyExact(    multiplyFull(
multiplyHigh(    negateExact(       nextAfter(        nextDown(
nextUp(          pow(               random()          rint(
round(           scalb(             signum(           sin(
sinh(            sqrt(              subtractExact(    tan(
tanh(            toDegrees(         toIntExact(       toRadians(
ulp(              unsignedMultiplyHigh(
jshell> Math.
```

```
jshell> Math.pow(
$1   $2

Signatures:
double Math.pow(double a, double b)

<press tab again to see documentation>
jshell> Math.pow(
```

```
jshell> Math.pow(
double Math.pow(double a, double b)
Returns the value of the first argument raised to the power of the
second argument. Special cases:
  * If the second argument is positive or negative zero, then the result
is 1.0.
...
<press tab again to see next page>
jshell> Math.pow(
```

```
<press tab again to see next page>
```

```
jshell> Math.PI
Signatures:
Math.PI:double

<press tab again to see documentation>
```

```
jshell> Math.PI
Math.PI:double
The double value that is closer than any other to pi (π), the ratio of
the circumference of a circle to its diameter.
jshell> Math.PI
```

```
jshell> Math
Math           MathContext

Signatures:
java.lang.Math

<press tab again to see documentation>
jshell> Math
```

```
jshell> Math
java.lang.Math
The class Math contains methods for performing basic numeric operations
such as the elementary exponential, logarithm, square root, and
trigonometric functions.
...

<press tab again to see next page>
jshell> Math
```

```
jshell> Math.abs(
$1   $2

Signatures:
int Math.abs(int a)
long Math.abs(long a)
float Math.abs(float a)
double Math.abs(double a)

<press tab again to see documentation>
jshell> Math.abs(
```

```
jshell> Math.abs(
int Math.abs(int a)
Returns the absolute value of an int value.If the argument is not
negative, the argument is returned. If the argument is negative, the
negation of the argument is returned.
...

<press tab again to see next page>
```

```
jshell> String dayName = "Monday"
dayName ==> "Monday"


jshell>
```

```
jshell> dayName.
charAt(                 chars()                 codePointAt(
codePointBefore(        codePointCount(         codePoints()
compareTo(              compareToIgnoreCase(    concat(
contains(               contentEquals(          describeConstable()
endsWith(               equals(                 equalsIgnoreCase(
formatted(              getBytes(               getChars(
getClass()              hashCode()              indent(
indexOf(                intern()                isBlank()
isEmpty()               lastIndexOf(            length()
lines()                 matches(                notify()
notifyAll()             offsetByCodePoints(     regionMatches(
repeat(                 replace(                replaceAll(
replaceFirst(           resolveConstantDesc(    split(
splitWithDelimiters(    startsWith(             strip()
stripIndent()           stripLeading()          stripTrailing()
subSequence(            substring(              toCharArray()
toLowerCase(            toString()              toUpperCase(
transform(              translateEscapes()      trim()
wait(
```

```
jshell> dayName.
```

```
jshell> dayName.toUpperCase(
Signatures:
String String.toUpperCase(Locale locale)
String String.toUpperCase()

<press tab again to see documentation>
jshell> dayName.toUpperCase(
```

```
jshell> $2.substring(
Signatures:
String String.substring(int beginIndex)
String String.substring(int beginIndex, int endIndex)

<press tab again to see documentation>
jshell> $2.substring(
```

```
jshell> $2.substring(
String String.substring(int beginIndex)
Returns a string that is a substring of this string.The substring begins
with the character at the specified index and extends to the end of this
string.
...
<press tab again to see next page>
```

```
void displayCubes() {
   for (int i = 1; i <= 10; ++i) {
      System.out.printf("Cube of %d is %d%n", i, cube(i));
   }
}
```

```
|  created method displayCubes(), however, it cannot be invoked until
method cube(int) is declared

jshell>

jshell> int[] values = {10, 20, 30}
values ==> int[3] { 10, 20, 30 }

jshell> values[1]
$2 ==> 20

jshell> values[10]
|  Exception java.lang.ArrayIndexOutOfBoundsException: Index 10 out of
bounds for length 3
|        at (#3:1)

jshell>
```

```
jshell> /imports
|    import java.io.*
|    import java.math.*
|    import java.net.*
|    import java.nio.file.*
|    import java.util.*
|    import java.util.concurrent.*
|    import java.util.function.*
|    import java.util.prefs.*
|    import java.util.regex.*
|    import java.util.stream.*

jshell>
```

```
jshell> /env -class-path libs/jackson-core-2.18.1.jar;libs/jackson-
databind-2.18.1.jar;libs/jackson-annotations-2.18.1.jar
|  Setting new options and restoring state.

jshell>

jshell> import com.fasterxml.jackson.databind.ObjectMapper;
jshell> /imports
|    import java.io.*
|    import java.math.*
|    import java.net.*
|    import java.nio.file.*
|    import java.util.*
|    import java.util.concurrent.*
|    import java.util.function.*
|    import java.util.prefs.*
|    import java.util.regex.*
|    import java.util.stream.*
|    import com.fasterxml.jackson.databind.ObjectMapper

jshell>
```

```
jshell> record Account(int accountNumber, String name, double balance) {}
|  created record Account

jshell> List<Account> accounts = List.of(
   ...>    new Account(100, "Devi", 24.98),
   ...>    new Account(200, "Taylor", 345.67));
accounts ==> [Account[accountNumber=100, name=Devi, balance=24 ...
e=Taylor, balance=345.67]]

jshell>

jshell> var mapper = new ObjectMapper();
mapper ==> com.fasterxml.jackson.databind.ObjectMapper@7e5afaa6

jshell> mapper.
<press tab again to see all possible completions; total possible
completions: 247>
jshell> mapper.
acceptJsonFormatVisitor(              activateDefaultTyping(
activateDefaultTypingAsProperty(      addHandler(
addMixIn(                             addMixInAnnotations(
...
jshell> mapper.

jshell> var prettyWriter = mapper.writerWithDefaultPrettyPrinter()
prettyWriter ==> com.fasterxml.jackson.databind.ObjectWriter@57d7f8ca

jshell>

jshell> System.out.println(prettyWriter.writeValueAsString(accounts))
[ {
  "accountNumber" : 100,
  "name" : "Devi",
  "balance" : 24.98
}, {
  "accountNumber" : 200,
  "name" : "Taylor",
  "balance" : 345.67
} ]

jshell>
```

```
jshell> /set editor notepad.exe
|  Editor set to: notepad.exe

jshell>
```

`/set editor -retain` *commandToLaunchYourEditor*

```
jshell> /list

   1 : 45
   2 : 72
   3 : if ($1 < $2) {
          System.out.printf("%d < %d%n", $1, $2);
       }
   4 : if ($1 > $2) {
          System.out.printf("%d > %d%n", $1, $2);
       }
   5 : $1 = 100;
   6 : if ($1 > $2) {
          System.out.printf("%d > %d%n", $1, $2);
       }

jshell>
```

```
jshell> /reload
|  Restarting and restoring state.
-: 45
-: 72
-: if ($1 < $2) {
      System.out.printf("%d < %d%n", $1, $2);
   }
45 < 72
-: if ($1 > $2) {
      System.out.printf("%d > %d%n", $1, $2);
   }
-: $1 = 100
-: if ($1 > $2) {
      System.out.printf("%d > %d%n", $1, $2);
   }
100 > 72

jshell>
```

```
jshell> /reload -quiet
|  Restarting and restoring state.
45 < 72
100 > 72

jshell>
```

```
jshell> int x = 10
x ==> 10

jshell> int cube(int y) {return y * y * y;}
|  created method cube(int)

jshell> /list

   1 : int x = 10;
   2 : int cube(int y) {return y * y * y;}

jshell> /drop 1
|  dropped variable x

jshell> /drop cube
|  dropped method cube(int)

jshell> /list

jshell>

jshell> /set feedback verbose
|  Feedback mode: verbose

jshell> int x = 10
x ==> 10
|  created variable x : int

jshell> int cube(int y) {return y * y * y;}
|  created method cube(int)

jshell> cube(x)
$3 ==> 1000
|  created scratch variable $3 : int

jshell> x = 5
x ==> 5
|  assigned to x : int

jshell> cube(x)
$5 ==> 125
|  created scratch variable $5 : int

jshell>
```

```
jshell> /set feedback concise
jshell> int x = 10
jshell> int cube(int y) {return y * y * y;}
jshell> cube(x)
$3 ==> 1000
jshell> x = 5
jshell> cube(x)
$5 ==> 125
jshell>
```

```
jshell> /set feedback silent
-> int x = 10
-> int cube(int y) {return y * y * y;}
-> cube(x)
-> x = 5
-> cube(x)
-> /set feedback normal
|  Feedback mode: normal

jshell>
```

```
jshell> System.out.println("Happy Birthday!");
Happy Birthday!

jshell>
```

```
jshell> System.out.println("Happy Birthday!")
Happy Birthday!

jshell>
```

```
jshell> // comments are not executable

jshell>
```

```
jshell> /* opening line of multi-line comment
   ...>    System.out.println("Welcome to Java Programming!")
   ...>    closing line of multi-line comment */

jshell>
```

```
/* incomplete multi-line comment
System.out.println(“Welcome to Java Programming!”)
/* complete multi-line
comment */
```

```
jshell> /* incomplete multi-line comment
   ...> System.out.println("Welcome to Java Programming!")
   ...> /* complete multi-line
   ...> comment */

jshell>

jshell> System.out.println("A")
A

jshell>    System.out.println("A") // indented 3 spaces
A

jshell>       System.out.println("A") // indented 6 spaces
A

jshell>

jshell> "Unmatched brace { in a string is OK"
$1 ==> "Unmatched brace { in a string is OK"

jshell>

jshell> System.out.println("seems OK")
seems OK

jshell> System.out.println("missing something?)
|  Error:
|  unclosed string literal
|  System.out.println("missing something?)
|                     ^

jshell> System.out.println"missing something else?")
|  Error:
|  ';' expected
|  System.out.println"missing something else?")
|                    ^
|  Error:
|  cannot find symbol
|    symbol:   variable println
|  System.out.println"missing something else?")
|  ^----------------^

jshell>
```

```
jshell> System.out.print("Happy ")
Happy
jshell> System.out.print("Birthday")
Birthday
jshell> /save mysnippets

jshell> /open mysnippets
Happy Birthday
jshell>
```

```
jshell> System.out.println("Happy ")
Happy

jshell> System.out.println("Birthday")
Birthday

jshell> /save mysnippets

jshell> /open mysnippets
Happy
Birthday

jshell>
```

```
jshell> System.out.println("Welcome\nto\nJShell!")
Welcome
to
JShell!

jshell>
```

```
"before\tafter\nbefore\t\tafter"
```

```
jshell> System.out.println("before\tafter\nbefore\t\tafter")
before  after
before          after

jshell>
```

```
jshell> System.out.println("Bad escap\e")
|  Error:
|  illegal escape character
|  System.out.println("Bad escap\e")
|                                  ^

jshell>

jshell> System.out.println("Displaying backslashes \\\\")
Displaying backslashes \\

jshell>

jshell> System.out.println("\"This is a string in quotes\"")
"This is a string in quotes"

jshell>

System.out.println("Happy Birthday!\rSunny")

jshell> System.out.println("Happy Birthday!\rSunny")
Sunny Birthday!

jshell>

System.out.printf("%s%n%s%n", "Welcome to ", "Java Programming!")


jshell> System.out.printf"%s%n%s%n", "Welcome to ", "Java Programming!"
|  Error:
|  ';' expected
|  System.out.printf"%s%n%s%n", "Welcome to ", "Java Programming!"
|                   ^
|  Error:
|  cannot find symbol
|    symbol:   variable printf
|  System.out.printf"%s%n%s%n", "Welcome to ", "Java Programming!"
|  ^---------------^

jshell>

jshell> System.out.printf("%s%n%s%n" "Welcome to " "Java Programming!")
|  Error:
|  ')' or ',' expected
|  System.out.printf("%s%n%s%n" "Welcome to " "Java Programming!")
|                               ^

jshell>

jshell> System.out.printf("%s%n", "Welcome to ", "Java Programming!")
Welcome to
$6 ==> java.io.PrintStream@2038ae61

jshell>
```

```
jshell> System.out.printf("%s%n%s%n", "Welcome to ")
Welcome to
|  Exception java.util.MissingFormatArgumentException: Format specifier
'%s'
|        at Formatter.format (Formatter.java:2800)
|        at PrintStream.implFormat (PrintStream.java:1369)
|        at PrintStream.format (PrintStream.java:1348)
|        at PrintStream.printf (PrintStream.java:1247)
|        at (#7:1)

jshell>

jshell> System.out.printf("%d%n%s%n", "Welcome to ", "Java Programming!")
|  Exception java.util.IllegalFormatConversionException: d !=
java.lang.String
|        at Formatter$FormatSpecifier.failConversion
(Formatter.java:4710)
|        at Formatter$FormatSpecifier.printInteger (Formatter.java:3248)
|        at Formatter$FormatSpecifier.print (Formatter.java:3203)
|        at Formatter.format (Formatter.java:2801)
|        at PrintStream.implFormat (PrintStream.java:1369)
|        at PrintStream.format (PrintStream.java:1348)
|        at PrintStream.printf (PrintStream.java:1247)
|        at (#8:1)

jshell>

jshell> System.out.printf("%s%n%s%n", 23, "Java Programming!")
23
Java Programming!
$9 ==> java.io.PrintStream@2038ae61

jshell>
```

```
jshell> /imports
|    import java.io.*
|    import java.math.*
|    import java.net.*
|    import java.nio.file.*
|    import java.util.*
|    import java.util.concurrent.*
|    import java.util.function.*
|    import java.util.prefs.*
|    import java.util.regex.*
|    import java.util.stream.*

jshell>
```

```
jshell> import java.util.Scanner

jshell> var input = new Scanner(System.in)
input ==> java.util.Scanner[delimiters=\p{javaWhitespace}+] ...
\E][infinity string=\Q∞\E]

jshell> int number = input.nextInt()
hello
|  Exception java.util.InputMismatchException
|        at Scanner.throwFor (Scanner.java:964)
|        at Scanner.next (Scanner.java:1619)
|        at Scanner.nextInt (Scanner.java:2284)
|        at Scanner.nextInt (Scanner.java:2238)
|        at (#12:1)
jshell>

jshell> var input = new Scanner(System.in)
input ==> java.util.Scanner[delimiters=\p{javaWhitespace}+] ...
\E][infinity string=\Q∞\E]

jshell> int number = input.nextInt()
hello
|  Exception java.util.InputMismatchException
|        at Scanner.throwFor (Scanner.java:964)
|        at Scanner.next (Scanner.java:1619)
|        at Scanner.nextInt (Scanner.java:2284)
|        at Scanner.nextInt (Scanner.java:2238)
|        at (#2:1)

jshell>
```

```
var input = new Scanner(System.in)
int value = input.nextInt()
```

```
jshell> var input = new Scanner(System.in)
input ==> java.util.Scanner[delimiters=\p{javaWhitespace}+] ...
\E][infinity string=\Q∞\E]

jshell> int value = input.nextInt()
```

```
jshell> class Demonstration {
   ...>    import java.util.Scanner;
   ...> }
|  Error:
|  illegal start of type
|     import java.util.Scanner;
|     ^
|  Error:
|  <identifier> expected
|     import java.util.Scanner;
|                             ^

jshell> import java.util.Scanner

jshell> class Demonstration {
   ...> }
|  created class Demonstration

jshell>
```

```
jshell> String month = "April"
month ==> "April"

jshell> month
month ==> "April"

jshell> int age = 65
age ==> 65

jshell> age
age ==> 65

jshell> /vars
|    String month = "April"
|    int age = 65

jshell>
```

```
jshell> 1234567890 == 1_234_567_890
$1 ==> true

jshell> System.out.println(1234567890)
1234567890

jshell> System.out.println(1_234_567_890)
1234567890

jshell> 1234567890 / 2
$4 ==> 617283945

jshell> 1_234_567_890 / 2
$5 ==> 617283945

jshell>
```

```
System.out.printf("Sum is %d%n", (number1 + number2))
```

```
jshell> int number1 = 10
number1 ==> 10

jshell> int number2 = 20
number2 ==> 20

jshell> System.out.printf("Sum is %d%n", (number1 + number2))
Sum is 30
$3 ==> java.io.PrintStream@23e028a9

jshell> System.out.printf("Sum is %d%n", number1 + number2)
Sum is 30
$4 ==> java.io.PrintStream@23e028a9

jshell>
```

```
jshell> 0 / 1
$1 ==> 0

jshell> 1 / 0
|  java.lang.ArithmeticException thrown: / by zero
|        at (#2:1)

jshell> 0 / 0
|  java.lang.ArithmeticException thrown: / by zero
|        at (#3:1)

jshell>
```

```
jshell> 5 / 2 * 2 + 4 % 3 + 9 - 3
$11 ==> 11

jshell>
```

```
if (3 == 5); {
   System.out.println("3 is equal to 5");
}
```

```
jshell> if (3 == 5); {
   ...>    System.out.println("3 is equal to 5");
   ...> }
3 is equal to 5

jshell>
```

```
jshell> (3 * 9 * (3 + (9 * 3 / (3))))
$42 ==> 324

jshell>
```

```java
// Fig. B.1: IntegerConversionTest.java
// Using the integer conversion characters.

public class IntegerConversionTest {
   public static void main(String[] args) {
      System.out.printf("%d%n", 26);
      System.out.printf("%d%n", +26);
      System.out.printf("%d%n", -26);
      System.out.printf("%o%n", 26);
      System.out.printf("%x%n", 26);
      System.out.printf("%X%n", 26);
   }
}
```

```
26
26
-26
32
1a
1A
```

printf(*format-string*, *argument-list*);

```java
// Fig. B.2: FloatingNumberTest.java
// Using floating-point conversion characters.

public class FloatingNumberTest {
   public static void main(String[] args) {
      System.out.printf("%e%n", 12345678.9);
      System.out.printf("%e%n", +12345678.9 );
      System.out.printf("%e%n", -12345678.9);
      System.out.printf("%E%n", 12345678.9);
      System.out.printf("%f%n", 12345678.9);
      System.out.printf("%g%n", 12345678.9);
      System.out.printf("%G%n", 12345678.9);
   }
}
```

```
1.234568e+07
1.234568e+07
-1.234568e+07
1.234568E+07
12345678.900000
1.23457e+07
1.23457E+07
```

```java
// Fig. B.3: CharStringConversion.java
// Using character and string conversion characters.
public class CharStringConversion {
   public static void main(String[] args) {
      char character = 'A';  // initialize char
      String string = "This is also a string";  // String object
      Integer integer = 1234;  // initialize integer (autoboxing)

      System.out.printf("%c%n", character);
      System.out.printf("%s%n", "This is a string");
      System.out.printf("%s%n", string);
      System.out.printf("%S%n", string);
      System.out.printf("%s%n", integer); // implicit call to toString
   }
}
```

```
A
This is a string
This is also a string
THIS IS ALSO A STRING
1234
```

```java
// Fig. B.4: OtherConversion.java
// Using the b, B, h, H, % and n conversion characters.

public class OtherConversion {
   public static void main(String[] args) {
      Object test = null;
      System.out.printf("%b%n", false);
      System.out.printf("%b%n", true);
      System.out.printf("%b%n", "Test");
      System.out.printf("%B%n", test);
      System.out.printf("Hashcode of \"hello\" is %h%n", "hello");
      System.out.printf("Hashcode of \"Hello\" is %h%n", "Hello");
      System.out.printf("Hashcode of null is %H%n", test);
      System.out.printf("Printing a %% in a format string%n");
      System.out.printf("Printing a new line %nnext line starts here");
   }
}
```

```
false
true
true
FALSE
Hashcode of "hello" is 5e918d2
Hashcode of "Hello" is 42628b2
Hashcode of null is NULL
Printing a % in a format string
Printing a new line
next line starts here
```

```java
// Fig. B.5: FieldWidthTest.java
// Right aligning integers in fields.

public class FieldWidthTest {
   public static void main(String[] args) {
      System.out.printf("%4d%n", 1);
      System.out.printf("%4d%n", 12);
      System.out.printf("%4d%n", 123);
      System.out.printf("%4d%n", 1234);
      System.out.printf("%4d%n%n", 12345); // data too large

      System.out.printf("%4d%n", -1);
      System.out.printf("%4d%n", -12);
      System.out.printf("%4d%n", -123);
      System.out.printf("%4d%n", -1234); // data too large
      System.out.printf("%4d%n", -12345); // data too large
   }
}
```

```
   1
  12
 123
1234
12345

  -1
 -12
-123
-1234
-12345
```

```java
// Fig. B.6: PrecisionTest.java
// Using precision for floating-point numbers and strings.
public class PrecisionTest {
   public static void main(String[] args) {
      double f = 123.94536;
      String s = "Happy Birthday";

      System.out.printf("Using precision for floating-point numbers%n");
      System.out.printf("\t%.3f%n\t%.3e%n\t%.3g%n%n", f, f, f);

      System.out.printf("Using precision for strings%n");
      System.out.printf("\t%.11s%n", s);
   }
}
```

```
Using precision for floating-point numbers
        123.945
        1.239e+02
        124

Using precision for strings
        Happy Birth
```

```java
// Fig. B.7: MinusFlagTest.java
// Right aligning and left aligning values.

public class MinusFlagTest {
   public static void main(String[] args) {
      System.out.println("Columns:");
      System.out.println("012345678901234567890123456789012345678901234567890123456789\n");
      System.out.printf("%10s%10d%10c%10f%n%n", "hello", 7, 'a', 1.23);
      System.out.printf(
         "%-10s%-10d%-10c%-10f%n", "hello", 7, 'a', 1.23);
   }
}
```

```
Columns:
012345678901234567890123456789012345678901234567890123456789

     hello         7         a  1.230000

hello     7         a         1.230000
```

```java
// Fig. B.8: PlusFlagTest.java
// Printing numbers with and without the + flag.

public class PlusFlagTest {
   public static void main(String[] args) {
      System.out.printf("%d\t%d%n", 786, -786);
      System.out.printf("%+d\t%+d%n", 786, -786);
   }
}
```

```
786     -786
+786    -786
```

```java
// Fig. B.9: SpaceFlagTest.java
// Printing a space before non-negative values.

public class SpaceFlagTest {
   public static void main(String[] args) {
      System.out.printf("% d%n% d%n", 547, -547);
   }
}
```

```
 547
-547
```

```java
// Fig. B.10: PoundFlagTest.java
// Using the # flag with conversion characters o and x.

public class PoundFlagTest {
   public static void main(String[] args) {
      int c = 31;       // initialize c

      System.out.printf("%#o%n", c);
      System.out.printf("%#x%n", c);
   }
}
```

```
037
0x1f
```

```java
// Fig. B.11: ZeroFlagTest.java
// Printing with the 0 (zero) flag fills in leading zeros.

public class ZeroFlagTest {
   public static void main(String[] args) {
      System.out.printf("%+09d%n", 452);
      System.out.printf("%09d%n", 452);
      System.out.printf("% 9d%n", 452);
   }
}
```

```
+00000452
000000452
      452
```

```java
// Fig. B.12: CommaFlagTest.java
// Using the comma (,) flag to display numbers with thousands separator.

public class CommaFlagTest {
   public static void main(String[] args) {
      System.out.printf("%,d%n", 58625);
      System.out.printf("%,.2f%n", 58625.21);
      System.out.printf("%,.2f%n", 12345678.9);
   }
}
```

```
58,625
58,625.21
12,345,678.90
```

```java
// Fig. B.13: ParenthesesFlagTest.java
// Using the ( flag to place parentheses around negative numbers.

public class ParenthesesFlagTest {
   public static void main(String[] args) {
      System.out.printf("%(d%n", 50);
      System.out.printf("%(d%n", -50);
      System.out.printf("%(.1e%n", -50.0);
   }
}
```

```
50
(50)
(5.0e+01)
```

```java
// Fig. B.14: ArgumentIndexTest.java
// Reordering output with argument indices.

public class ArgumentIndexTest {
   public static void main(String[] args) {
      System.out.printf(
         "Parameter list without reordering: %s %s %s %s%n",
         "first", "second", "third", "fourth");
      System.out.printf(
         "Parameter list after reordering: %4$s %3$s %2$s %1$s%n",
         "first", "second", "third", "fourth");
   }
}
```

```
Parameter list without reordering: first second third fourth
Parameter list after reordering: fourth third second first
```

```java
// Fig. B.15: FormatterTest.java
// Formatting output with class Formatter.
import java.util.Formatter;

public class FormatterTest {
   public static void main(String[] args) {
      // create Formatter and format output
      Formatter formatter = new Formatter();
      formatter.format("%d = %#o = %#X", 10, 10, 10);

      System.out.println(formatter.toString());
   }
}
```

```
10 = 012 = 0XA
```

https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/util/Formatter.html

```java
String s = String.format("%d = %#o = %#x", 10, 10, 10);
```

```java
String s = "%d = %#o = %#x".formatted(10, 10, 10);
```

```java
String book = """
      Title: "%s"
    Edition: %d
  Copyright: %d""".formatted("Java How to Program", 12, 2025);
```

Binary number Octal equivalent Hexadecimal equivalent
100011010001 4321 8D1

Positional values: 64 32 16 8 4 2 1

Positional values: 32 16 8 4 2 1

Positional values: 32 16 8 4 2 1
Symbol values: 1 1 1 0 0 1

Positional values: 512 64 8 1

Positional values: 64 8 1

Positional values: 64 8 1
Symbol values: 1 4 7

Positional values: 4096 256 16 1

Positional values: 256 16 1

Positional values: 256 16 1
Symbol values: 1 7 7

00000000 00000000 00000000 00001101

onesComplementOfValue = ~value;

value:
00000000 00000000 00000000 00001101

~value (i.e., value's one's complement):
11111111 11111111 11111111 11110010

Two's complement of value:
11111111 11111111 11111111 11110011

```
 00000000 00000000 00000000 00001101
+11111111 11111111 11111111 11110011
-----------------------------------
 00000000 00000000 00000000 00000000
```

```
a (i.e., 27)          00000000 00000000 00000000 00011011
+(~value + 1)        +11111111 11111111 11111111 11110011
                     -----------------------------------
                      00000000 00000000 00000000 00001110
```